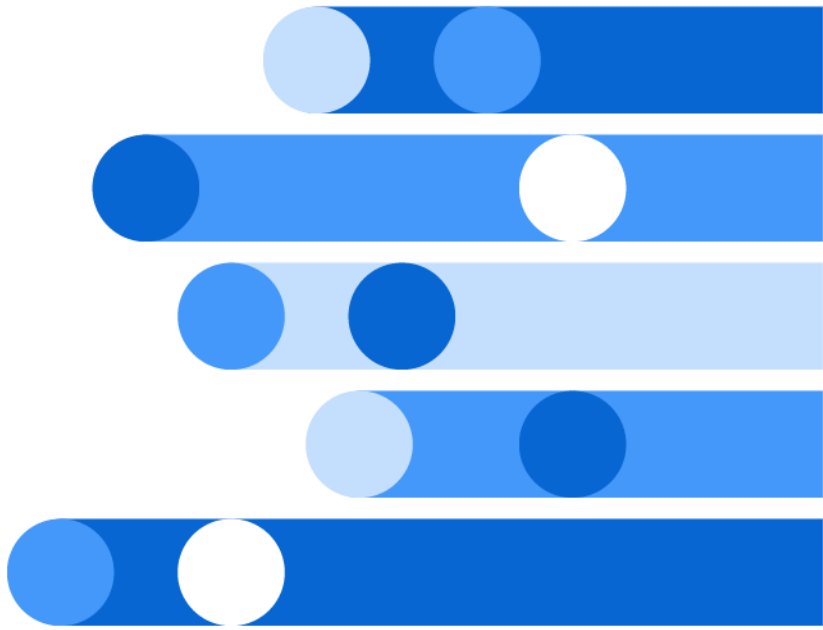
SAS®コンポーネントオブジェクト: リファレンス

2020.1 - 2025.03*



* このドキュメントは、ソフトウェアの追加バージョンに適用される場合があります。このドキュメントを SAS Help Center で開き、バナーのバージョンをクリックすると、使用できるすべてのバージョンが表示されます。

SAS® ドキュメント
2026 年 9 月 18 日

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2020. *SAS®コンポーネントオブジェクト: リファレンス*. Cary, NC: SAS Institute Inc.

SAS®コンポーネントオブジェクト: リファレンス

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

March 2025

v_001-P1:lecompobjref

目次

SAS 言語の構文規則 v

1 部 コンポーネントオブジェクトについて 1

1 章 / SAS コンポーネントオブジェクトについて 3
コンポーネントオブジェクト 3
コンポーネントインターフェイス 4
ドット表記とコンポーネントオブジェクト 5
コンポーネントオブジェクト使用時のヒント 6

2 部 DATA ステップコンポーネントオブジェクト 11

2 章 / ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用 13
ハッシュオブジェクトの使用 13
ハッシュ反復子オブジェクトの使用 21

3 章 / ハッシュオブジェクトとハッシュ反復子オブジェクト言語要素のディクショナリ 25
ディクショナリ 26

4 章 / Java オブジェクトの使用 105
Java オブジェクトの使用 105

5 章 / Java オブジェクトの言語要素のディクショナリ 125
カテゴリ別の Java オブジェクトメソッド 125
ディクショナリ 127

3 部 FCMP コンポーネントオブジェクト 155

6 章 / PROC FCMP ハッシュオブジェクトとハッシュ反復子オブジェクトの使用 157
PROC FCMP ハッシュオブジェクトと PROC FCMP ハッシュ反復子オブジェクトの使用 157

7 章 / PROC FCMP ハッシュおよびハッシュ反復子言語要素 159
ディクショナリ 160

8 章 / ディクショナリオブジェクトの使用 **219**
FCMP ディクショナリオブジェクトの使用 219

9 章 / ディクショナリオブジェクトの言語要素 **227**
ディクショナリ 227

10 章 / Python オブジェクトの使用 **251**
PROC FCMP Python オブジェクトの使用 251

11 章 / Python オブジェクトの言語要素 **263**
ディクショナリ 263

SAS 言語の構文規則

SAS 言語の構文規則

SAS 言語の構文規則の概要

SAS 言語の構文規則の概要

SAS 言語要素の構文のドキュメントでは、標準の規則が使用されています。これらの規則により、SAS 構文の構成要素を簡単に識別できます。規則は、次の項目に分類されます。

- 構文の構成要素
- 書体に関する規則
- 特殊文字
- SAS ライブラリや外部ファイルへの参照

構文の構成要素

構文の構成要素

ほとんどの言語要素の構文のコンポーネントには、キーワードと引数があります。キーワードのみ必要な言語要素もあります。また、一部の言語要素では、キーワードの後に等号(=)を付加する必要があります。複数の引数を含む構文で区切り記号を使用する場合と使用しない場合を説明するために、引数の構文の形式が複数示されています。

keyword
プログラムを記述するときに使用する SAS 言語要素の名前を指定します。キーワードはリテラルであり、通常、構文の先頭の単語です。CALL ルーチンでは、最初の 2 つの単語がキーワードです。

これらの例の SAS 構文では、キーワードには太字が使用されています。

CHAR (*string, position*)

CALL RANBIN (*seed, n, p, x*);

ALTER (*alter-password*)

BEST *w*.

REMOVE <*data-set-name*>

この例では、CALL ルーチンの最初の 2 つの単語がキーワードです。

CALL RANBIN(*seed*, *n*, *p*, *x*)

一部の SAS ステートメントの構文は、引数のない 1 つのキーワードで構成されます。

DO;
... *SAS code* ...
END;

一部のシステムオプションでは、2 つあるキーワード値から 1 つを指定する必要があります。

DUPLEX | **NODUPLEX**

プロシジャステートメントによっては、ステートメント構文中に複数のキーワードが含まれます。

CREATE <UNIQUE> **INDEX** *index-name* **ON** *table-name* (*column-1* <, *column-2*, ...>)

引数
数値または文字の定数、変数、式のいずれかを指定します。引数は、キーワードまたはキーワードの後の等号記号の後に続きます。SAS では、言語要素を処理するために引数が使用されます。引数は必須の場合と、省略可能な場合があります。構文では、オプションの引数は山かっこ(< >)で囲みます。

この例では、*string* と *position* がキーワード CHAR に続きます。これらの引数は、CHAR 関数の必須引数です。

CHAR (*string, position*)

各引数には値があります。この例の SAS コードでは、引数 *string* の値は 'summer'、引数 *position* の値は 4 です。

```
x=char('summer', 4);
```

この例では、*string* および *substring* は必須引数ですが、*modifiers* と *startpos* はオプションです。

FIND(*string, substring* <, *modifiers*> <, *startpos*>

argument(s)
引数は必ず 1 つ必要であり、複数の引数が許可されます。引数の間はスペースで区切ります。カンマ(,)などの区切り記号は、引数間に必要ありません。

たとえば、MISSING ステートメントは、この形式で複数の引数を含みます。

MISSING *character(s)*;

<LITERAL_ARGUMENT> *argument-1* <<LITERAL_ARGUMENT> *argument-2* ... >
引数は必ず 1 つ必要であり、リテラル引数がこの引数に関連付けられます。リテラルと引数のペアは複数指定できます。リテラルと引数の間に区切り記号は必要ありません。省略記号(...)は、追加のリテラルと引数が許可されることを示します。

たとえば、BY ステートメントはこの引数を含みます。

BY <DESCENDING> *variable-1* <<DESCENDING> *variable-2* ...>;

argument-1 <options> <argument-2 <options> ...>
引数は必ず 1 つ必要であり、1 つ以上のオプションがこの引数に関連付けられます。複数の引数と関連するオプションを指定できます。引数とオプションの間に区切り記号は必要ありません。省略記号(...)は、追加の引数と関連するオプションが許可されることを示します。

たとえば、FORMAT プロシジャの PICTURE ステートメントは、この形式で複数の引数を含みます。

PICTURE name <(*format-options*)>
<*value-range-set-1* <(*picture-1-options*)>
<*value-range-set-2* <(*picture-2-options*)> ...>>;

argument-1=value-1 <argument-2=value-2 ...>
引数には値を割り当てる必要があり、複数の引数を指定できます。省略記号(...)は、追加の引数が許可されることを示します。引数間に区切り記号は必要ありません。

たとえば、LABEL ステートメントは、この形式で複数の引数を含みます。

LABEL *variable-1=label-1 <variable-2=label-2 ...>*;

argument-1 <, *argument-2*, ...>
引数は必ず 1 つ必要であり、カンマまたは別の区切り記号で区切って複数の引数を指定できます。省略記号(...)は、カンマで区切られた引数が続くことを示します。SAS ドキュメントでは両方の形式が使用されます。

次に、この形式で指定された複数の引数の例を示します。

AUTHPROVIDERDOMAIN (*provider-1:domain-1* <, *provider-2:domain-2*, ...>

INTO :*macro-variable-specification-1* <, :*macro-variable-specification-2*, ...>

注: 通常、SAS ドキュメントのサンプルコードは、小文字の固定幅フォントを使用して表記されます。コードの作成には、大文字も、小文字も、大文字と小文字の両方も使用できます。

スタイル規則

書体に関する規則

SAS 構文の説明に使用されるスタイル規則には、大文字太字、大文字、斜体の規則も含まれます。

UPPERCASE BOLD
関数またはステートメントの名前など、SAS キーワードを示します。この例では、キーワード ERROR の表記には大文字太字が使用されています。

ERROR <*message*>;

UPPERCASE
リテラルである引数を表します。

この CMPMODEL=システムオプションの例では、BOTH、CATALOG、XML がリテラルです。

CMPMODEL=BOTH | CATALOG | XML |

italic
ユーザー指定の引数または値を示します。斜体表記の項目は、ユーザー指定値であり、次のいずれかを表します。

- 非リテラル引数。この LINK ステートメントの例では、引数 *label* はユーザー指定値のため、斜体で表示されます。

  LINK *label*;

- 引数に割り当てられる非リテラル値。

  この FORMAT ステートメントの例では、引数 DEFAULT に変数の *default-format* が割り当てられます。

  FORMAT *variable(s)* <*format*> <DEFAULT = *default-format*>;

特殊文字

特殊文字

SAS 言語要素の構文には、次の特殊文字を使用できます。

=
等号記号は、システムオプションなどの一部の言語要素内のリテラルの値を示します。

この MAPS システムオプションの例では、等号により MAPS の値が設定されます。

MAPS=*location-of-maps*

< >
山かっこは省略可能な引数を示します。必須引数は山かっこで囲みません。

この CAT 関数の例では、少なくとも項目が 1 つ必要です。

CAT (*item-1* <, *item-2*, ...>)

|
縦棒は、値グループから 1 つの値を選択できることを示します。縦棒で区切られている値は相互に排他的です。

この CMPMODEL=システムオプションの例では、引数を 1 つのみ選択できます。

CMPMODEL=BOTH | CATALOG | XML

...
省略記号は、引数の繰り返しが可能なことを示します。引数と省略記号が山かっこで囲まれている場合、その引数はオプションです。繰り返される引数には、その引数の前や後ろに、区切り記号を入れる必要があります。

この CAT 関数の例では、複数の *item* 引数が許可され、カンマで区切る必要があります。

CAT (*item-1 <, item-2*, ...>)

'*value*' or "*value*"
単一引用符や二重引用符付きの引数は、その値にも単一引用符または二重引用符を付ける必要があることを示します。

この FOOTNOTE ステートメントの例では、引数 *text* に引用符が付けられています。

FOOTNOTE <*n*> <*ods-format-options* '*text*' | "*text*">;

;
セミコロンは、ステートメントまたは CALL ルーチンの終了を示します。

この例では、各ステートメントがセミコロンで終了しています。

```
data namegame;
  length color name $8;
  color = 'black';
  name = 'jack';
  game = trim(color) || name;
run;
```

SAS ライブラリと外部ファイルへの参照

SAS ライブラリと外部ファイルへの参照

SAS ステートメントおよびその他の言語要素の多くは、SAS ライブラリと外部ファイルを参照します。論理名(ライブラリ参照名またはファイル参照名)から参照を作成するのか、引用符付きの物理ファイル名を使用するかを選択できます。

論理名を使用する場合、通常、参照の作成に SAS ステートメント(LIBNAME または FILENAME)を使用するのか、動作環境のコントロール言語を使用するのかを選択します。SAS ライブラリと外部ファイルを参照する方法は複数あり、一部の方法は動作環境によって異なります。

SAS ドキュメント中の外部ファイルを使用する例では、*file-specification* という語句を斜体で使用しています。また、SAS ライブラリを使用する例には斜体フレーズ *SAS-library* を引用符で囲んで使用します。

infile *file-specification* obs = 100;
libname *libref* '*SAS-library*';

1 部

コンポーネントオブジェクトについて

1 章
SAS コンポーネントオブジェクトについて 3

1

SAS コンポーネントオブジェクトについて

コンポーネントオブジェクト 3
コンポーネントインターフェイス 4
ドット表記とコンポーネントオブジェクト 5
定義 5
構文 5
コンポーネントオブジェクト使用時のヒント 6

コンポーネントオブジェクト

コンポーネントオブジェクトインターフェイスでは、DATA ステップや PROC FCMP で定義済みのコンポーネントオブジェクトを作成し、操作することができます。

SAS では、DATA ステップで使用するために事前定義された、次のコンポーネントオブジェクトが用意されています。

ハッシュオブジェクトとハッシュ反復子オブジェクト
: ルックアップキーに基づいて、データをすばやく効率的に保存、検索および取得できます。詳細については、2 章, “ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用”を参照してください

Java オブジェクト
: Java クラスをインスタンス化して結果オブジェクトのフィールドとメソッドにアクセスする、Java ネイティブインターフェイス (JNI)に似たメカニズムを提供します。Java オブジェクトの詳細については、“Java オブジェクトの使用”を参照してください。

ロガーおよびアペンダーオブジェクト
: ログイベントを記録し、それらのイベントを適切な出力先に書き込むことができます。詳細については、 xisError - link not found - The element n0g6bb7xpm747nn16s4u8517topl was not found in the link database を参照してください。

SAS では、PROC FCMP で使用するために事前定義された、次のコンポーネントオブジェクトが用意されています。

ディクショナリオブジェクト
: ディクショナリオブジェクトは、PROC FCMP プログラム内のデータを参照または値別に保存するための、別のより広い方法を提供します。詳細については、“FCMP ディクショナリオブジェクトの使用”を参照してください。

ハッシュオブジェクトとハッシュ反復子オブジェクト
: ルックアップキーに基づいて、データをすばやく効率的に保存、検索および取得できます。詳細については、“PROC FCMP ハッシュオブジェクトと PROC FCMP ハッシュ反復子オブジェクトの使用”を参照してください。

Python オブジェクト
: SAS から Python インタープリターに Python 関数をサブミットして実行することができます。詳細については、“PROC FCMP Python オブジェクトの使用” (251 ページ)を参照してください。

コンポーネントインターフェイス

コンポーネントオブジェクトを宣言作成するには、DECLARE ステートメント単体か、DECLARE ステートメントと_NEW_演算子を併用します。

コンポーネントオブジェクトは、属性、メソッド、演算子で構成されるデータ要素です。**属性**とは、オブジェクトに関連付けられた情報を指定するプロパティです。**メソッド**は、オブジェクトが実行できる演算を定義します。コンポーネントオブジェクトに対して、**演算子**は特別な機能を提供します。

コンポーネントオブジェクトの属性とメソッドにアクセスするには、オブジェクトのドット表記を使用します。

注: コンポーネントオブジェクトのステートメント、属性、メソッド、演算子は、これらのオブジェクト用に定義されている場合のみ使用できます。これらの事前定義された DATA ステップと PROCFCMP オブジェクトで SAS Component Language 機能を使用することはできません。

ドット表記とコンポーネントオブジェクト

定義

ドット表記は、メソッド呼び出しおよび属性値の設定とクエリに対するショートカットとして使用できます。ドット表記を使用することで、SAS プログラムが読みやすくなります。

DATA ステップコンポーネントオブジェクトでドット表記を使用するには、DECLARE ステートメントのみを使用するか、DECLARE ステートメントと_NEW_演算子を組み合わせて使用し、コンポーネントオブジェクトを宣言してインスタンス化する必要があります。PROC FCMP コンポーネントオブジェクトでは、DECLARE ステートメントのみを使用してください。および SAS Viya Logging Facility for SAS Programs and Jobs.

構文

ドット表記の構文を次に示します。

object.attribute

または

object.method (<*argument_tag-1*: *value-1* <,... *argument_tag-n*: *value-n*>>;

引数は次のように定義されます。

object
コンポーネントオブジェクトの変数名を指定します。

attribute
割り当てまたはクエリ用のオブジェクト属性を指定します。

オブジェクトの属性を設定する場合、コードの形式は次のようになります。

```
object.attribute = value;
```

オブジェクトの属性をクエリする場合、コードの形式は次のようになります。

```
value = object.attribute;
```

method
呼び出すメソッド名を指定します。

argument_tag
メソッドに渡される引数を識別します。引数タグはかっこで囲みます。メソッドに引数タグが含まれるかどうかに関わらず、かっこは必須です。

コンポーネントオブジェクトのメソッドは次のような形式になります。

```
return_code=object.method(<argument_tag-1:value-1
  <, ...argument_tag-n:value-n>>);
```

リターンコードは、メソッドの成功または失敗を示します。ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

value
引数の値を指定します。

コンポーネントオブジェクト使用時のヒント

- 変数の割り当てと同じ方法でオブジェクトを割り当てることができます。ただし、オブジェクトの種類は一致する必要があります。最初のコードセットは有効ですが、2 番目のコードセットではエラーが発生します。

```
declare hash h();
declare hash t();
t=h;
```

```
declare hash t();
declare javaobj j();
j=t;
```

- オブジェクトの配列は宣言できません。次のコードではエラーが生成されます。

```
declare hash h1();
declare hash h2();
array h h1–h2;
```

- コンポーネントオブジェクトはハッシュオブジェクト内にデータとして保存できます。

```
data _null_;
  declare hash h1();
  declare hash h2;     /* h2 is declared, but not instantiated here */

  length key1 key2 $20;

  /* This sets up h1 to have a character key, with key (key1)
    and object data (h2) */
  h1.defineKey('key1');
  h1.defineData('key1', 'h2');
  h1.defineDone();
```

```
  /* Add a record to an instance of h2 with key = 'xyz' */
  key2 = 'xyz';
  h2 = _new_ hash();
  h2.defineKey('key2');
  h2.defineDone();
  h2.add();

  /* Add a record to h1 with key = 'abc', and data equal to 'abc'
    and the current h2 */
  key1 = 'abc';
  h1.add();

  /* Create an instance of h2. Add another record to h1 with a
    different key and another instance of h2 as data */
  key2 = 'pqr';
  h2 = _new_ hash();
  h2.definekey('key2');
  h2.definedone();
  h2.add();
  key1 = 'def';
  h1.add();

  /* Retrieve the object in h1 with key1 */
  key1 = 'abc';
  rc = h1.find();
  h2.output(dataset: 'work.h1');

  /* Retrieve another object stored in h1 */
  key1 = 'def';
  rc = h1.find();
  h2.output(dataset: 'work.h2');
run;

proc print data=work.h1; run;
proc print data=work.h2; run;
```

データセット WORK.H1 を次に示します。

Obs	key2
1	xyz

データセット WORK.H2 を次に示します。

Obs	key2
1	pqr

- コンポーネントオブジェクトはハッシュオブジェクト内にキーとしては保存できません。次のコードではエラーが生成されます。

```
declare hash h();
length k 8;
h.definekey('k');
```

```
h.definedone();

declare hash h2();
h2.definekey('h');   /* Error */
h2.definedone();
```

- 等号記号(=)以外の比較演算子を含むコンポーネントオブジェクトは使用できません。H1 と H2 がハッシュオブジェクトの場合、次のコードではエラーが生成されます。

```
if h1>h2 then
```

- コンポーネントオブジェクトを宣言してインスタンス化した後に、スカラー値を割り当てることはできません。J が Java オブジェクトの場合、次のコードではエラーが生成されます。

```
j=5;
```

- まだ使用されている可能性のあるオブジェクト参照、または参照によってすでに削除されているオブジェクト参照を削除しないように注意する必要があります。次のコードでは、元の H1 オブジェクトは H2 への参照によってすでに削除されているため、2 番目の DELETE ステートメントでエラーが生成されます。元の H2 は直接参照できなくなります。

```
declare hash h1();
declare hash h2();
declare hash t();
t=h2;
h2=h1;
h2.delete();
t.delete();
```

- 引数タグの構文ではコンポーネントオブジェクトを使用できません。次の例では、ADD メソッドでの H2 ハッシュオブジェクトの使用でエラーが生成されます。

```
declare hash h2();
declare hash h();
h.add(key: h2);
h.add(key: 99, data: h2);
```

- Java によって書かれる SAS ログへのテキストの 1 バイト目でのパーセント文字(%)の使用は、SAS によって予約されています。Java テキスト行の 1 バイト目に%を書く必要がある場合は、そのすぐ後に別のパーセントを追加してエスケープします(%%)。
- ハッシュテーブルのハッシュテーブルを 持つこともできます。
- Java オブジェクトは単一 Java クラスのインスタンス化を表します。Java オブジェクトは他のものを持つことができません。しかし、Java インスタンスは他の通常の Java インスタンスと同様に任意に複雑化できます。Java オブジェクトは他の Java エンティティの参照を持つことができますが、それらは Java オブジェクトとは考慮されません。
- SAS がロックダウン状態の時、Java オブジェクトは使用できません。詳細については、“LOCKDOWN” (*SAS プログラマガイド: エッセンシャル*)を参照してください。

- コンポーネントオブジェクトは、In-database DATA ステップではサポートされていません。詳細については、“DATA Step Acceleration” (*SAS In-Database Products: User's Guide*)を参照してください。

2 部

DATA ステップコンポーネントオブジェクト

2 章
ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用 13

3 章
ハッシュオブジェクトとハッシュ反復子オブジェクト言語要素のディクショナリ 25

4 章
Java オブジェクトの使用 105

5 章
Java オブジェクトの言語要素のディクショナリ 125

2

ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用

ハッシュオブジェクトの使用 **13**
ハッシュオブジェクトを使用する理由 13
ハッシュオブジェクトの作成 14
コンストラクターを使ったハッシュオブジェクトデータの初期化 15
キーとデータの定義 15
キー値とデータのペアの重複 16
データの格納と取得 17
キー集計の維持 18
ハッシュオブジェクト属性の使用 20
ハッシュ反復子オブジェクトの使用 **21**
ハッシュ反復子オブジェクトについて 21
ハッシュ反復子オブジェクトの宣言とインスタンス化 21
例: ハッシュ反復子を使用してハッシュオブジェクトデータを取得する 22

ハッシュオブジェクトの使用

ハッシュオブジェクトを使用する理由

ハッシュオブジェクトは、迅速なデータの格納と取得のための効率的かつ便利なインメモリメカニズムを提供します。ハッシュオブジェクトは、ルックアップキーに基づいてデータを保存および取得します。

DATA ステップコンポーネントオブジェクトインターフェイスを使用するには、次の手順にしたがいます。

1 ハッシュオブジェクトを宣言します。

2 ハッシュオブジェクトのインスタンスを作成(インスタンス化)します。

3 ルックアップキーとデータを初期化します。

ハッシュオブジェクトを宣言してインスタンス化した後に、次を始めとする、多くのタスクを実行できます。

- データの保存と取得
- キー集計の維持
- データの置換と削除
- ハッシュオブジェクトの比較
- ハッシュオブジェクトのデータを含むデータセットの生成

たとえば、重複しない患者の番号と体重に対応する、数値の検査結果を含む大規模なデータセットがあるとします。そして、患者番号を含む小さなデータセット（大きなデータセットのサブセット）があるとします。重複しない患者番号をキーに使用し、体重の値をデータに使用して、ハッシュオブジェクトに大規模なデータセットをロードできます。患者番号を使用してシングルパスが小規模なデータセット上で作成され、体重が一定の値を上回る現在の患者がハッシュオブジェクトで検索し、そのデータを異なるデータセットに出力します。

ルックアップキーの数とデータセットのサイズによりますが、ハッシュオブジェクトのルックアップは、標準形式のルックアップよりも大幅に高速化されます。キーを検索しているだけで、メモリがたくさんあり、高速なパフォーマンスを求める場合は、大規模データセットを最初に読み込みます。大量のメモリを使用しないようにする場合は、小規模なデータセットを最初にロードします。

ハッシュオブジェクトの作成

DECLARE ステートメントを使用して、ハッシュオブジェクトを作成します。新しいハッシュオブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化します。例:

```
declare hash myhash;
myhash = _new_ hash();
```

DECLARE ステートメントは、オブジェクト参照 myhash がハッシュ型であることをコンパイラに示します。この時点では、オブジェクト参照 myhash のみを宣言しています。ハッシュ型のコンポーネントオブジェクトを保持する可能性があります。ハッシュオブジェクトの宣言は一度のみ行います。_NEW_演算子でハッシュオブジェクトのインスタンスが作成され、オブジェクト参照 myhash に割り当てられます。

DECLARE ステートメントと_NEW_演算子を使用してコンポーネントオブジェクトを宣言してインスタンス化する 2 ステップのプロセス以外の方法もあります。DECLARE ステートメントを使用して、ハッシュオブジェクトの宣言とインスタンス化を 1 つのステップで実行できます。

```
declare hash myhash();
```

上記のステートメントは、次のコードと同等です。

```
declare hash myhash;
myhash = _new_ hash();
```

詳細については、“DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)と“_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)を参照してください。

コンストラクターを使ったハッシュオブジェクトデータの初期化

ハッシュオブジェクトを作成する場合、コンストラクターは、ハッシュオブジェクトをインスタンス化し、ハッシュオブジェクトデータを初期化するために使用できるメソッドです。

ハッシュオブジェクトのコンストラクターは、次のいずれかの形式になります。

- declare hash *object_name*(*argument_tag-1*: *value-1*
  <, ...*argument_tag-n*: *value-n*>);
- *object_name* = _new_ hash(*argument_tag-1*: *value-1*
  <, ...*argument_tag-n*: *value-n*>);

詳細については、“DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)と“_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)を参照してください。

キーとデータの定義

ハッシュオブジェクトはルックアップキーを使用して、データの保存と取得を行います。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。すべてのキー変数とデータ変数を定義した後、DEFINEDONE メソッドが呼び出されます。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

例えば、次のコードでは、文字キーと文字データ変数を初期化しています。

```
length d $20;
length k $20;

if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k');
  rc = h.defineData('d');
  rc = h.defineDone();
end;
```

複数のキー変数とデータ変数を保有できますが、MULTIDATA:“YES”引数タグ付きのハッシュオブジェクトを作成しない場合はキーが重複しない必要があります。詳細については、“キー値とデータのペアの重複”を参照してください。

特定の 1 つのキーを使用して、複数のデータアイテムを保存できます。たとえば、前の例を変更して、補助数値を文字キーと文字データと一緒に保存できるようにできます。この例では、キーとデータアイテムはそれぞれ、文字値と数値で構成されます。

```
length d1 8;
length d2 $20;
length k1 $20;
length k2 8;

if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k1', 'k2');
  rc = h.defineData('d1', 'd2');
  rc = h.defineDone();
end;
```

詳細については、“DEFINEDATA メソッド” (42 ページ)、“DEFINEDONE メソッド” (44 ページ)、“DEFINEKEY メソッド” (46 ページ)を参照してください。

注: ハッシュオブジェクトは値をキー変数(例:**h.find(key:'abc')**)に割り当てないため、SAS コンパイラはハッシュオブジェクトとハッシュ反復子が実行するデータ変数割り当てを検出できません。したがって、キー変数またはデータ変数への割り当てがプログラムに出現しない場合、変数が初期化されていないことを示す NOTE が発行されます。このような NOTE が発行されないようにするには、次のアクションのいずれかを実行します。

- NONOTES システムオプションを設定します。
- 各キー変数およびデータ変数について、初期割り当てステートメントを(通常は欠損値に)指定します。
- CALL MISSING ルーチンを、すべてのキー変数とデータ変数をパラメーターとして使用します。次に例を示します。

```
length d $20;
length k $20;

if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k');
  rc = h.defineData('d');
  rc = h.defineDone();
  call missing(k, d);
end;
```

キー値とデータのペアの重複

デフォルトでは、各キーに対して 1 セットのデータ変数が存在します。しかし、状況によっては、ハッシュオブジェクトのキーを重複させる必要があるかもしれません(つまり、1 つのキーに複数のデータ変数のセットを関連付けます)。

たとえば、キーが患者 ID であり、データは来院日であるとします。患者が複数回来院した場合、複数の日付が患者 ID に関連付けられます。MULTIDATA:"YES"引数タグ付きのハッシュオブジェクトを作成すると、データ変数の複数のセットが 1 つのキーに関連付けられます。

データセットに重複するキーが含まれている場合、デフォルトでは、最初のインスタンスがハッシュオブジェクトに保存され、次のインスタンスは無視されます。ハッシュオブジェクトに最後のインスタンスを保存するには、DUPLICATE 引数タグを使用します。また、DUPLICATE 引数タグは、重複したキーが存在する場合に SAS ログにエラーを書き込みます。

しかし、DECLARE ステートメントまたは_NEW_演算子に MULTIDATA 引数タグを使用する場合、ハッシュオブジェクトでは各キーに複数の値を保存できます。ハッシュオブジェクトは、キーに関連付けられているリストに複数の値を保持します。このリストは、HAS_NEXT や FIND_NEXT などの複数のメソッドを使用して移動や操作を行えます。

複数のデータアイテムを含むリスト内を移動するには、現在のリストアイテムを確認する必要があります。開始するには、キーを指定して FIND メソッドを呼び出します。FIND メソッドによって、現在のリストアイテムが設定されます。キーに複数のデータ値があるかを確認するには、HAS_NEXT メソッドを呼び出します。キーが別のデータ値を持つことを確認した後、FIND_NEXT メソッドでその値を取得できます。FIND_NEXT メソッドは、現在のリストアイテムをリスト内の次のアイテムに設定し、そのアイテムのデータ変数を設定します。

指定されたキーのリストを前方に移動する以外に、同じように HAS_PREV メソッドと FIND_PREV メソッドを使用して、後方にリストをループできます。

ハッシュオブジェクトが単一のキーに対して複数の値を持つ場合、DO_OVER メソッドを DO ループの反復内で使用して、重複キー間を移動します。DO_OVER メソッドは最初のメソッド呼び出しでキーを読み込み、キーが最後に到達するまで重複キーリストを移動しつづけます。

注: 複数のデータアイテムを含むリストのアイテムは、挿入順序で保持されます。

重複キーとデータのペアに関連付けられたこれらのメソッドや他のメソッドの詳細については、"ハッシュオブジェクトとハッシュ反復子オブジェクト言語要素のディクショナリ" (25 ページ)を参照してください。

データの格納と取得

データの格納方法と取得方法

ハッシュオブジェクトのキー変数とデータ変数を初期化した後に、ADD メソッドを使用してハッシュオブジェクトにデータを保存したり、*dataset* 引数タグを使用してハッシュオブジェクトにデータセットをロードしたりできます。*dataset* 引数タグを使用する際にデータセットに同じキー値を持つ複数のオブザベーションが含まれている場合、デフォルトでは、SAS はハッシュテーブル内の最初のオブザベーションを保持し、次のオブザベーションは無視します。重複するキーがある場合に、ハ

ッシュオブジェクトの最後のインスタンスを保存するか、またはログにエラーを書き込むには、DUPLICATE 引数タグを使用します。キーの重複値を許可するには、MULTIDATA 引数タグを使用します。

次に、キーに 1 つのデータ値が存在する場合、FIND メソッドを使用して、ハッシュオブジェクトからデータを検索して取得できます。キーに複数のデータアイテムが存在する場合、FIND_NEXT メソッドと FIND_PREV メソッドを使用して、データを検索して取得します。

詳細については、“ADD メソッド” (26 ページ)、“FIND メソッド” (53 ページ)、“FIND_NEXT メソッド” (55 ページ)、および“FIND_PREV メソッド” (57 ページ)を参照してください。

REF メソッドを使用して、FIND メソッドと ADD メソッドを連結できます。次の例では、コード量をここから

```
rc = h.find();
 if (rc != 0) then
   rc = h.add();
```

1 つのメソッド呼び出しに縮約できます。

```
rc = h.ref();
```

詳細については、“ハッシュ反復子オブジェクトの使用 ” (21 ページ)を参照してください。

注: また、ハッシュ反復子オブジェクトを使用し、ハッシュオブジェクトのデータを一度に 1 つのデータアイテムづつ、前方または後方の順で取得できます。詳細については、“データの格納方法と取得方法”を参照してください。

キー集計の維持

ハッシュオブジェクトの宣言時に SUMINC 引数タグを使用すると、ハッシュオブジェクトキーの集計数を維持できます。タグ値は、数値 DATA ステップ変数(SUMINC 変数)の名前に解決される文字列式です。

この SUMINC タグによって、ハッシュオブジェクトに各キーの集計値を維持する内部ストレージを割り当てるように示されます。

ADD メソッドまたは REPLACE メソッドが使用されると、常にハッシュキーの集計値は初期化され、SUMINC 変数の値にされます。

FIND メソッド、CHECK メソッド、または REF メソッドが使用されると、常にハッシュキーの集計値に SUMINC 変数の値が加算されます。

SUMINC 変数は、負、正、またはゼロに評価されることがあります。変数は整数である必要はありません。キーの SUMINC 値は、デフォルトではゼロです。

次の例では、ADD の前に、最初の ADD メソッドによって K=99 の集計数が 1 に設定されます。新しい COUNT 値が指定されるたびに、次の FIND メソッドは値をキー集計に加算します。この例では、キーごとに 1 つのデータ値が存在します。SUM メソッドは、キー集計の現在値を取得します。値は、DATA ステップ変数 TOTAL に保

存されます。各キーに複数のアイテムが存在する場合、SUMDUP メソッドはキー集計の現在値を取得します。

```
data _null_;
 length k count 8;
 length total 8;
 dcl hash myhash(suminc: 'count');
 myhash.defineKey('k');
 myhash.defineDone();

 k = 99;
 count = 1;
 myhash.add();


/* COUNT is given the value 2.5 and the */
/* FIND sets the summary to 3.5*/
 count = 2.5;
 myhash.find();

/* The COUNT of 3 is added to the FIND and */
/* sets the summary to 6.5. */
 count = 3;
 myhash.find();


/* The COUNT of -1 sets the summary to 5.5. */
 count = -1;
 myhash.find();

/* The SUM method gives the current value of */
/* the key summary to the variable TOTAL. */
 myhash.sum(sum: total);

/* The PUT statement prints total=5.5 in the log. */
 put total=;
 run;
```

この例では、各キー値 K=99 と K=100 の集計が維持されます。

```
 k = 99;
 count = 1;
 myhash.add();
 /* key=99 summary is now 1 */

 k = 100;
 myhash.add();
 /* key=100 summary is now 1 */

 k = 99;
 myhash.find();
 /* key=99 summary is now 2 */

 count = 2;
 myhash.find();
 /* key=99 summary is now 4 */

 k = 100;
```

```
myhash.find();
/* key=100 summary is now 3 */

myhash.sum(sum: total);
put 'total for key 100 = 'total;

k = 99;

myhash.sum(sum:total);
put 'total for key 99 = ' total;
```

最初の PUT ステートメントは K=100 の集計を出力します。

```
total for key 100 = 3
```

2 番目の PUT ステートメントは K=99 の集計を出力します。

```
total for key 99 = 4
```

キー集計は、*dataset* 引数タグと組み合わせて使用できます。DEFINEDONE メソッドを使用してデータセットをハッシュオブジェクトに読み込み、すべてのキー集計を SUMINC 値に設定します。すべての後続の FIND メソッド、CHECK メソッド、または ADD メソッドは、対応するキー集計を変更します。

```
declare hash myhash(suminc: "keycount", dataset: "work.mydata");
```

ハッシュオブジェクト属性の使用

DATA ステップコンポーネントインターフェイスでは、属性を使用してハッシュオブジェクトから情報を取得できます。属性には次の構文を使用します。

```
attribute_value=obj.attribute_name;
```

ハッシュオブジェクトで使用できる属性は 2 つあります。NUM_ITEMS はハッシュオブジェクトのアイテム数を返し、ITEM_SIZE はアイテムのサイズ(バイト)を返します。次の例では、ハッシュオブジェクトのアイテム数を取得します。

```
n = myhash.num_items;
```

次の例では、ハッシュオブジェクトのアイテムのサイズを 取得します。

```
s = myhash.item_size;
```

ハッシュオブジェクトが ITEM_SIZE 属性と NUM_ITEMS 属性でどれだけのメモリ量を使用しているかという考え方を取得できます。ITEM_SIZE 属性は、ハッシュオブジェクトが必要とする初期オーバーヘッドを反映せず、必要な内部配置も考慮しません。そのため、ITEM_SIZE を使用した場合は正確なメモリ使用量は提供されず、近似値が指定されます。

詳細については、“NUM_ITEMS 属性” (74 ページ)と“ITEM_SIZE 属性” (64 ページ)を参照してください。

ハッシュ反復子オブジェクトの使用

ハッシュ反復子オブジェクトについて

ハッシュ反復子オブジェクトを使用して、ルックアップキーに基づきデータを保存して検索します。ハッシュ反復子オブジェクトにより、ハッシュオブジェクトのデータを前向きまたは逆向きのキー順序で取得できます。

ハッシュ反復子オブジェクトの宣言とインスタンス化

DECLARE ステートメントを使用して、ハッシュ反復子オブジェクトを宣言します。新しいハッシュ反復子オブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化します。引数タグとしてハッシュオブジェクト名を使用します。例:

```
declare hiter myiter;
myiter = _new_ hiter('h');
```

DECLARE ステートメントは、オブジェクト参照 MyIter の種類がハッシュ反復子であることをコンパイラに示します。この時点では、オブジェクト参照である MyIter のみを宣言しています。ハッシュ反復子型のコンポーネントオブジェクトを保持する可能性があります。ハッシュ反復子オブジェクトの宣言は一度のみ行います。_NEW_演算子でハッシュ反復子オブジェクトのインスタンスが作成され、オブジェクト参照 MyIter に割り当てられます。ハッシュオブジェクト H がコンストラクター引数として渡されます。つまり、ハッシュオブジェクト変数ではなく、ハッシュオブジェクトがハッシュ反復子に割り当てられます。

DECLARE ステートメントと_NEW_演算子を使用してコンポーネントオブジェクトの宣言とインスタンス化を行う 2 ステップのプロセス以外に、コンストラクターメソッドとして DECLARE ステートメントを使用して、1 ステップでハッシュ反復子オブジェクトの宣言とインスタンス化を行えます。構文は次の通りです。

declare hiter *object_name*(*hash_object_name*);

上記の例では、ハッシュオブジェクト名は、単一引用符または二重引用符で囲む必要があります。

例:

```
declare hiter myiter('h');
```

前の文ステートメントはこれらに相当します。

```
declare hiter myiter;
```

```
myiter = _new_ hiter('h');
```

注: ハッシュオブジェクトを宣言してインスタンス化してから、ハッシュ反復子オブジェクトを作成する必要があります。詳細については、"ハッシュオブジェクトの作成"を参照してください。

例:

```
if _N_ = 1 then do;
  length key $10;
  declare hash myhash(dataset:"work.x", ordered: 'yes');
  declare hiter myiter('myhash');
  myhash.defineKey('key');
  myhash.defineDone();
end;
```

このコードは、変数名 MyIter を使用して、ハッシュ反復子オブジェクトのインスタンスを作成します。ハッシュオブジェクト MyHash がコンストラクター引数として渡されます。ORDERED 引数タグを**'yes'**に設定してハッシュオブジェクトを作成したため、データは昇順の key-value で返されます。

DECLARE ステートメントと_NEW_演算子の詳細については、*SAS DATA ステップステートメント: リファレンス*を参照してください。

例: ハッシュ反復子を使用してハッシュオブジェクトデータを取得する

次のコードは天文データを含むデータセット ASTRO を使用して、赤経(RA)の値が 12 より大きいメシエ天体を含むデータセット（OBJ）を作成します。FIRST メソッドと NEXT メソッドを使用して、昇順にデータを取得します。FIRST メソッドと NEXT メソッドの詳細については、*SAS コンポーネントオブジェクト: リファレンス*を参照してください。

```
data astro;
  input obj $1-4 ra $6-12 dec $14-19;
  datalines;
M31 00 42.7 +41 16
M71 19 53.8 +18 47
M51 13 29.9 +47 12
M98 12 13.8 +14 54
M13 16 41.7 +36 28
M39 21 32.2 +48 26
M81 09 55.6 +69 04
M100 12 22.9 +15 49
M41 06 46.0 -20 44
M44 08 40.1 +19 59
M10 16 57.1 -04 06
M57 18 53.6 +33 02
 M3 13 42.2 +28 23
M22 18 36.4 -23 54
M23 17 56.8 -19 01
M49 12 29.8 +08 00
```

```
 M68 12 39.5 -26 45
 M17 18 20.8 -16 11
 M14 17 37.6 -03 15
 M29 20 23.9 +38 32
 M34 02 42.0 +42 47
 M82 09 55.8 +69 41
 M59 12 42.0 +11 39
 M74 01 36.7 +15 47
 M25 18 31.6 -19 15
;
run;

data out;
  if _N_ = 1 then do;
    length obj $10;
    length ra $10;
    length dec $10;
    /* Read ASTRO data set and store in asc order in hash obj */
    declare hash h(dataset:"work.astro", ordered: 'yes');
    /* Define variables RA and OBJ as key and data for hash object */
    declare hiter iter('h');
    h.defineKey('ra');
    h.defineData('ra', 'obj');
    h.defineDone();
    /* Avoid uninitialized variable notes */
    call missing(obj, ra, dec);
  end;
/* Retrieve RA values in ascending order */
rc = iter.first();
do while (rc = 0);
/* Find hash object keys greater than 12 and output data */
  if ra GE '12' then
    output;
  rc = iter.next();
end;
run;

proc print data=work.out;
  var ra obj;
  title 'Messier Objects Greater than 12 Sorted by Right Ascension Values';
run;
```

次の出力は、PROC PRINT が生成するレポートを示します。

Messier Objects Greater than 12 Sorted by Right Ascension Values

Obs	ra	obj
1	12 13.8	M98
2	12 22.9	M100
3	12 29.8	M49
4	12 39.5	M68
5	12 42.0	M59
6	13 29.9	M51
7	13 42.2	M3
8	16 41.7	M13
9	16 57.1	M10
10	17 37.6	M14
11	17 56.8	M23
12	18 20.8	M17
13	18 31.6	M25
14	18 36.4	M22
15	18 53.6	M57
16	19 53.8	M71
17	20 23.9	M29
18	21 32.2	M39

3

ハッシュオブジェクトとハッシュ反復子オブジェクト言語要素のディクショナリ

ディクショナリ **26**
ADD メソッド 26
CHECK メソッド 28
CLEAR メソッド 30
DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト 32
DEFINEDATA メソッド 42
DEFINEDONE メソッド 44
DEFINEKEY メソッド 46
DELETE メソッド: ハッシュオブジェクトとハッシュ反復子オブジェクト 48
DO_OVER メソッド 49
EQUALS メソッド 51
FIND メソッド 53
FIND_NEXT メソッド 55
FIND_PREV メソッド 57
FIRST メソッド 58
HAS_NEXT メソッド 61
HAS_PREV メソッド 63
ITEM_SIZE 属性 64
LAST メソッド 65
NEW 演算子: ハッシュとハッシュ反復子 67
NEXT メソッド 72
NUM_ITEMS 属性 74
OUTPUT メソッド 75
PREV メソッド 80
REF メソッド 81
REMOVE メソッド 84
REMOVEDUP メソッド 87
REPLACE メソッド 89
REPLACEDUP メソッド 93
RESET_DUP メソッド 96

SETCUR メソッド 97
SUM メソッド 99
SUMDUP メソッド 101

ディクショナリ

ADD メソッド

特定のキーに関連付けられた指定データをハッシュオブジェクトに追加します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**ADD** (<<KEY: *keyvalue-1*>, ...<KEY: *keyvalue-n*>,
<DATA: *datavalue-1* >, ... <DATA: *datavalue-n*>>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 *keyvalue* にコンポーネントオブジェクトを含めることはできません。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

"DATA: *datavalue*"ペアの数は、DEFINEDATA メソッドを使用して定義されたデータ変数の数によって変わります。

制限事項 *datavalue* にコンポーネントオブジェクトを含めることはできません。

詳細

次の 2 つの方法のいずれかで ADD メソッドを使用して、ハッシュオブジェクト内にデータを保存できます。

1 つ目は、次のコードに示すように、キーおよびデータアイテムを 定義してから ADD メソッドを使用する方法です。

```
data _null_;
  length k $8;
  length d $12;
  /* Declare hash object and key and data variable names */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
  end;
  rc = h.output(dataset:'beforeadd');
  /* Define constant key and data values */
  k = 'Joyce';
  d = 'Ulysses';
  /* Add key and data values to hash object */
  rc = h.add();
  rc = h.output(dataset:'afteradd');
run;

proc print data = beforedadd; run;
proc print data = afteradd; run;
```

このエラーは、ハッシュにデータを追加する前に BEFOREADD データセットを表示しようとしたときにログに書き込まれます。

```
ERROR: File WORK.BEFOREDADD.DATA does not exist.
```

キーとデータの値を定義すると、ハッシュ内のデータを表示することができます。

Obs	d
1	Ulysses

2 つ目は、次のコードに示すように、ショートカットを使用して、ADD メソッド呼び出しでキーおよびデータアイテムを 直接指定する方法です。

```
data _null_;
  length k $8;
  length d $12;
  /* Define hash object and key and data variable names */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
```

```
    call missing(k, d);
  end;
  rc = h.output(dataset:'beforeadd');
  /* Define constant key and data values and add to hash object */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  rc = h.output(dataset:'afteradd');
run;

proc print data = beforedadd; run;
proc print data = afteradd ; run;
```

すでにハッシュオブジェクト内にあるキーを追加した場合、ADD メソッドはゼロ以外の値を返します。指定したキーに関連付けられたデータアイテムを新しいデータに置き換えるには、REPLACE メソッドを使用します。

DEFINEDATA メソッドでデータ変数を指定しない場合、データ変数は自動的にキーと同じであるとみなされます。

KEY および DATA 引数タグを使用して直接キーとデータを指定する場合、両方の引数タグを使用する必要があります。

ADD メソッドでは、データ変数の値はデータアイテムの値に設定されません。ADD メソッドはハッシュオブジェクトの値のみを設定します。キー変数またはデータ変数への割り当てがこのプログラムに出現しないため、変数が初期化されていないことを示す NOTE が発行されます。これらの NOTE を受け取らないようにするには、CALL MISSING ルーチンを使用して、キーとデータ変数をパラメーターとして指定します。

関連項目

- “データの格納と取得”

メソッド:

- “DEFINEDATA メソッド” (42 ページ)
- “DEFINEKEY メソッド” (46 ページ)
- “OUTPUT メソッド” (75 ページ)
- “REF メソッド” (81 ページ)

CHECK メソッド

指定したキーがハッシュオブジェクト内に保存されているかどうかを確認します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**CHECK** (<KEY: *keyvalue-1*, ... KEY: *keyvalue-n*>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで CHECK メソッドを使用して、ハッシュオブジェクト内のデータを検索できます。

1 つ目は、次のコードに示すように、キーを指定してから CHECK メソッドを使用する方法です。

```
data _null_;
  length k $8;
  length d $12;
  /* Declare hash object and key and data variable names */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();

 /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
  /* Define constant key and data values and add to hash object */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  /* Verify that JOYCE key is in hash object */
  k = 'Joyce';
  rc = h.check();
  if (rc = 0) then
    put 'Key is in the hash object.';
run;
```

2 つ目は、次のコードに示すように、ショートカットを使用して、CHECK メソッド呼び出しでキーを直接指定する方法です。

```
data _null_;
  length k $8;
  length d $12;
  /* Declare hash object and key and data variable names */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();

 /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
  /* Define constant key and data values and add to hash object */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  /* Verify that JOYCE key is in hash object */
  rc = h.check(key: 'Joyce');
  if (rc  =0) then
    put 'Key is in the hash object.';
run;
```

比較

CHECK メソッドは、キーがハッシュオブジェクト内にあるかどうかを示す値のみを返します。キーに関連付けられたデータ変数は更新されません。FIND メソッドも、キーがハッシュオブジェクト内にあるかどうかを示す値を返します。ただし、キーがハッシュオブジェクト内にある場合、FIND メソッドはさらにデータ変数にデータアイテムの値を設定し、メソッド呼び出し後にそのデータアイテムを使用できるようにします。

関連項目

メソッド:

- “DEFINEKEY メソッド” (46 ページ)
- “FIND メソッド” (53 ページ)

CLEAR メソッド

ハッシュオブジェクトインスタンスを削除せずに、ハッシュオブジェクトからすべてのアイテムを 削除します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**CLEAR** ();

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

詳細

CLEAR メソッドでは、既存のハッシュオブジェクトを削除して新しいオブジェクトを作成することなく、そのオブジェクトからアイテムを削除して再利用できます。ハッシュオブジェクトインスタンスを完全に削除する場合は、DELETE メソッドを使用します。

注: CLEAR メソッドは、DATA ステップ変数の値を変更しません。ハッシュオブジェクトの値のみをクリアします。

例: ハッシュオブジェクトのクリア

次の例では、ハッシュオブジェクトを宣言し、ハッシュオブジェクト内のアイテム数を取得してから、ハッシュオブジェクトを削除せずにクリアします。

```
data mydata;
  do i = 1 to 10000;
    output;
  end;
run;
data _null_;
  length i 8;

/* Declares the hash object named MYHASH using the data set MyData. */
dcl hash myhash(dataset: 'mydata');
 myhash.definekey('i');
 myhash.definedone();
 call missing (i);
/* Uses the NUM_ITEMS attribute, which returns the */
/* number of items in the hash object.            */
 n = myhash.num_items;
 put n=;
/* Uses the CLEAR method to delete all items within MYHASH. */
```

```
 rc = myhash.clear();
/* Writes the number of items in the log. */
 n = myhash.num_items;
 put n=;
 run;
```

最初の PUT ステートメントは、ハッシュテーブル MYHASH をクリアする前にそのアイテム数を書き込みます。

```
n=10000
```

2 番目の PUT ステートメントは、ハッシュテーブル MYHASH をクリアした後にそのアイテム数を書き込みます。

```
n=0
```

関連項目

メソッド:

- “DELETE メソッド: ハッシュオブジェクトとハッシュ反復子オブジェクト” (48 ページ)

DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト

ハッシュオブジェクトまたはハッシュ反復子オブジェクトを宣言します。ハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンスを作成し、データを初期化します。

該当要素:	DATA ステップ
カテゴリ:	アクション
種類:	実行
別名:	DCL
適用対象:	ハッシュオブジェクト, ハッシュ反復子オブジェクト

構文

形式 1: **DECLARE** *object object-reference*;

形式 2: **DECLARE** *object object-reference* (<*argument_tag-1*: *value-1*, ...*argument_tag-n*: *value-n*>);

引数

object
コンポーネントオブジェクトを指定します。次のいずれかの値を使用できます。

hash
ハッシュオブジェクトを指定します。ハッシュオブジェクトは、迅速なデータの保存および取得のメカニズムを提供します。ハッシュオブジェクトは、ルックアップキーに基づいてデータを保存および取得します。

参照項目: “データの格納と取得”

hiter
ハッシュ反復子オブジェクトを指定します。ハッシュ反復子オブジェクトを使用すると、ハッシュオブジェクトのデータを正または逆のキー順序で取得できます。

参照項目: “データの格納と取得”

object-reference
ハッシュオブジェクトまたはハッシュ反復子オブジェクトのオブジェクト参照名を指定します。

argument_tag:*value*
ハッシュオブジェクトのインスタンスの作成に使用する情報を指定します。

有効なハッシュオブジェクト引数および値タグには、次の 7 つがあります。

dataset: '*dataset_name* <(*datasetoption*)>'

注: ハッシュオブジェクトは、データセット名に対しての VALIDMEMNAME=EXTEND システムオプションをサポートしていません。データセット名には、特殊文字や各国語文字は使用できません。

ハッシュオブジェクトに読み込む SAS データセットの名前を指定します。

SAS データセットの名前には、リテラルまたは文字変数を使用できます。リテラルデータセット名は単一引用符または二重引用符で囲む必要があります。データセットの名前を含む文字変数は、引用符で囲んではいけません。マクロ変数は二重引用符で囲む必要があります。

> 重要 ハッシュオブジェクトが CAS で処理される場合、SAS データセットオプションはサポートされません。

DATASET 引数タグでハッシュオブジェクトを宣言するときに、SAS データセットオプションを使用できます。データセットオプションは、それらが表示される SAS データセットにのみ適用されるアクションを指定し、次の操作を実行できるようにします。

- 変数名を変更する
- 処理するオブザベーション番号に基づくオブザベーションのサブセットの選択
- WHERE オプションを使用したオブザベーションの選択

- ハッシュオブジェクトに読み込まれたデータセット、または OUTPUT メソッド呼び出しで指定されている出力データセットの変数の削除または保持
- データセットのパスワードを指定する

次の構文が使用されます。

```
dcl hash h (dataset: 'x (where = (i > 10))');
```

SAS データセットオプションのリストについては、*SAS データセットオプション: リファレンス*を参照してください

注 データセットに重複するキーが含まれている場合、デフォルトでは、ハッシュオブジェクトの最初のインスタンスがロードされ、後続のインスタンスは無視されます。DUPLICATE 引数タグを使用して、最後のインスタンスをハッシュオブジェクトに格納するか、SAS ログにエラーメッセージを書き込みます。1 つのキーに対して複数のデータアイテムを 許可するには、MULTIDATA 引数タグを使用します。

duplicate: '*option*'

重複したキーが許されていない場合の処理方法を決定します。デフォルトでは、最初のキーが保存され、後続の重複は無視されます。Option には次のいずれかの値を指定できます。

'replace'
'r'
最後の重複するキーレコードを保存します。

'error'
'e'
重複するキーが検出された場合に、ログにエラーを報告します。

REPLACE オプションを使用する次の例では、 *color_id* キー 531 には **blue**、*color_id* キー 620 には **brown** が保存されます。デフォルトでは、最初の値 (531 は **yellow**、620 は **green**)がロードされ、重複する値は無視されます。

```
data table;
 input color_id color_name $;
 datalines;
 531 yellow
 620 green
 531 blue
 908 orange
 620 brown
 143 purple
run;

data _null_;
length color_id 8 color_name $ 8;
if (_n_ = 1) then do;
   declare hash myhash(dataset: "table", duplicate: "r");
   rc = myhash.definekey('color_id');
   rc = myhash.definedata('color_id','color_name');
   myhash.definedone();
   /* avoid uninitialized variable notes */
   call missing(color_id, color_name);
end;
```

```
rc = myhash.output(dataset:"otable");
run;
```

hashexp: *n*
ハッシュオブジェクトの内部テーブルサイズです。ハッシュテーブルのサイズは 2^n です。

HASHEXP の値は、ハッシュテーブルサイズを作成する 2 の累乗指数として使用されます。たとえば、HASHEXP の値 4 は、ハッシュテーブルサイズ 2^4(つまり 16)と同じです。HASHEXP の最大値は 20 です。

ハッシュテーブルサイズは、保存可能なアイテム数とは等しくありません。ハッシュテーブルを、一連の'バケット'と考えます。ハッシュテーブルサイズ 16 には、16 個の'バケット'があります。各バケットに保持できるアイテムの数は無限です。ハッシュテーブルの効率は、アイテムをバケットにマップし、バケットからアイテムを取得するハッシュ関数の機能にあります。

ハッシュオブジェクトのルックアップルーチンの効率を最大にするには、ハッシュオブジェクトのデータ量に応じてハッシュテーブルサイズを指定する必要があります。最適な結果が得られるまで、さまざまな HASHEXP 値を試してみてください。たとえば、ハッシュオブジェクトに 100 万個のアイテムが含まれている場合、ハッシュテーブルサイズ 16(HASHEXP = 4)でも機能しますが、効率は良くありません。ハッシュテーブルサイズ 512 または 1024(HASHEXP = 9 または 10)を使用すると、最高の処理速度が得られます。

デフォルト　8。これは、ハッシュテーブルサイズ 2^8(つまり 256)と同じです。

keysum:'*variable-name*'
すべてのキーのキー集計をトラッキングする変数の名前を指定します。キー集計はキーが何回 FIND メソッド呼び出しに参照されたかのカウントです。

注　キー集計は出力データセットにあります。

例　“例 5: 出力データセットにキー集計を追加する” (41 ページ)

multidata: '*option*'
各キーに複数のデータアイテムを許可するかどうかを指定します。

option には、次のいずれかの値を使用できます。

'YES'
'Y'
　各キーに複数のデータアイテムが許可されます。

'NO'
'N'
　各キーにデータアイテムが 1 つのみ許可されます。

デフォルト　NO

ヒント　引数値は二重引用符で囲むこともできます。

参照項目:　“キー値とデータのペアの重複”

ordered: '*option*'
ハッシュオブジェクトとハッシュ反復子オブジェクトを併用する場合、またはハッシュオブジェクト OUTPUT メソッドを使用する場合、データがキー値の順序で返されるかどうか、またはどのように返されるかを指定します。

option には、次のいずれかの値を使用できます。

'ascending'
'a'
データはキー値の昇順で返されます。'**ascending**'の指定は、'**yes**'を指定することと同じです。

'descending'
'd'
データはキー値の降順で返されます。

'YES'
'Y'
データはキー値の昇順で返されます。'**yes**'の指定は、'**ascending**'を指定することと同じです。

'NO'
'N'
データは未定義の順序で返されます。

デフォルト NO

ヒント 引数は二重引用符で囲むこともできます。

suminc: '*variable-name*'
ハッシュオブジェクトキーの集計数を維持します。SUMINC 引数タグは、DATA ステップ変数を指定します。この変数には、合計増分が保持されます。合計増分はキーが参照されるたびにキー集計に追加される数です。

参照項目: "キー集計の維持"

例 キー集計は、DATA ステップ変数の現在の値を使用して変更されます。

```
dcl hash myhash(suminc: 'count');
```

参照項目: "コンストラクターを使ったハッシュオブジェクトデータの初期化""ハッシュ反復子オブジェクトの宣言とインスタンス化"

詳細

基本情報

SAS プログラムで DATA ステップコンポーネントオブジェクトを使用するには、オブジェクトを宣言して作成(インスタンス化)する必要があります。定義済みのハッシュおよびハッシュ反復子コンポーネントオブジェクトには、DATA ステップコンポーネントインターフェイスからアクセスできます。

定義済みの DATA ステップコンポーネントオブジェクトの詳細については、2 章, “ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用”を参照してください。

ハッシュオブジェクトまたはハッシュ反復子オブジェクトの宣言(フォーム 1)

ハッシュオブジェクトまたはハッシュ反復子オブジェクトを宣言するには、DECLARE ステートメントを使用します。

```
declare hash h;
```

DECLARE ステートメントは、オブジェクト参照 H がハッシュオブジェクトであることを SAS に示します。

新しいハッシュオブジェクトまたはハッシュ反復子オブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化します。たとえば、次のコード行では、_NEW_演算子でハッシュオブジェクトが作成され、オブジェクト参照 H に割り当てられます。

```
h = _new_ hash( );
```

DECLARE ステートメントを使用したハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンス生成(フォーム 2)

DECLARE ステートメントと_NEW_演算子を使用してハッシュオブジェクトまたはハッシュ反復子オブジェクトを宣言し、インスタンス化する 2 ステップのプロセスの代わりに、DECLARE ステートメントを使用して、ハッシュオブジェクトまたはハッシュ反復子オブジェクトを 1 つのステップで宣言し、インスタンス化することができます。たとえば、次のコード行では、DECLARE ステートメントでハッシュオブジェクトが宣言およびインスタンス化され、オブジェクト参照 H に割り当てられます。

```
declare hash h( );
```

前述のコード行は、次のコードを使用するのと同等です。

```
declare hash h;
h = _new_ hash( );
```

constructor は、ハッシュオブジェクトをインスタンス化し、ハッシュオブジェクトデータを初期化するために使用できるメソッドです。たとえば、次のコード行では、DECLARE ステートメントでハッシュオブジェクトが宣言およびインスタンス化され、オブジェクト参照 H に割り当てられます。さらに、引数タグ HASHEXP を使用してハッシュテーブルサイズが値 16(2^4)に初期化されます。

```
declare hash h(hashexp: 4);
```

ハッシュオブジェクトの読み込み時に SAS データセットオプションを使用する

> 重要　ハッシュオブジェクトが CAS で処理される場合、SAS データセットオプションはサポートされません。

DATASET 引数タグでハッシュオブジェクトを宣言するときに、SAS データセットオプションを使用できます。データセットオプションは、それらが表示される SAS データセットにのみ適用されるアクションを指定し、次の操作を実行できるようにします。

- 変数名を変更する
- 処理するオブザベーション番号に基づくオブザベーションのサブセットの選択
- WHERE オプションを使用したオブザベーションの選択
- ハッシュオブジェクトに読み込まれたデータセット、または OUTPUT メソッド呼び出しで指定されている出力データセットの変数の削除または保持
- データセットのパスワードを指定する

次の構文が使用されます。

```
dcl hash h(dataset: 'x (where = (i > 10))');
```

データセットオプションのその他の使用例については、“例 4: ハッシュオブジェクトの読み込み時に SAS データセットオプションを使用する” (40 ページ)を参照してください。データセットオプションのリストについては、*SAS データセットオプション: リファレンス*を参照してください。

比較

ハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンスを宣言し、インスタンス化するには、DECLARE ステートメントと_NEW_演算子を使用するか、DECLARE ステートメントのみを使用します。

例

例 1: DECLARE ステートメントと_NEW_演算子を使用したハッシュオブジェクトの宣言とインスタンス生成

この例では、DECLARE ステートメントを使用してハッシュオブジェクトを宣言します。ハッシュオブジェクトのインスタンス化には_NEW_演算子が使用されます。

```
data _null_;
  length k $15;
  length d $15;
  if _N_ = 1 then do;
    /* Declare and instantiate hash object "myhash" */
    declare hash myhash;
    myhash = _new_ hash( );
    /* Define key and data variables */
    rc = myhash.defineKey('k');
    rc = myhash.defineData('d');
    rc = myhash.defineDone( );
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
```

```
  /* Create constant key and data values */
  rc = myhash.add(key: 'Labrador', data: 'Retriever');
  rc = myhash.add(key: 'Airedale', data: 'Terrier');
  rc = myhash.add(key: 'Standard', data: 'Poodle');
  /* Find data associated with key and write data to log */
  rc = myhash.find(key: 'Airedale');
  if (rc = 0) then
    put d=;
  else
    put 'Key Airedale not found';
run;
```

例 2: DECLARE ステートメントを使用したハッシュオブジェクトの宣言とインスタンス生成

この例では、DECLARE ステートメントを使用してハッシュオブジェクトの宣言とインスタンス化を 1 つのステップで実行します。

```
data _null_;
  length k $15;
  length d $15;
  if _N_ = 1 then do;
    /* Declare and instantiate hash object "myhash" */
    declare hash myhash( );
    rc = myhash.defineKey('k');
    rc = myhash.defineData('d');
    rc = myhash.defineDone( );
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;

  /* Create constant key and data values */
  rc = myhash.add(key: 'Labrador', data: 'Retriever');
  rc = myhash.add(key: 'Airedale', data: 'Terrier');
  rc = myhash.add(key: 'Standard', data: 'Poodle');
  /* Find data associated with key and write data to log*/
  rc = myhash.find(key: 'Airedale');
  if (rc = 0) then
    put d=;
  else
    put 'Key Airedale not found';
run;
```

例 3: ハッシュオブジェクトのインスタンスを生成してサイズ順に並べる

この例では、DECLARE ステートメントを使用してハッシュオブジェクトを宣言し、インスタンス化します。ハッシュテーブルサイズは 16(2^4)に設定されています。

```
data _null_;
  length k $15;
  length d $15;
  if _N_ = 1 then do;
    /* Declare and instantiate hash object "myhash". */
    /* Set hash table size to 16. */
```

```
    declare hash myhash(hashexp: 4);
    rc = myhash.defineKey('k');
    rc = myhash.defineData('d');
    rc = myhash.defineDone( );
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;

  /* Create constant key and data values */
  rc = myhash.add(key: 'Labrador', data: 'Retriever');
  rc = myhash.add(key: 'Airedale', data: 'Terrier');
  rc = myhash.add(key: 'Standard', data: 'Poodle');
  rc = myhash.find(key: 'Airedale');
  /* Find data associated with key and write data to log*/
  if (rc = 0) then
    put d=;
  else
    put 'Key Airedale not found';
run;
```

例 4: ハッシュオブジェクトの読み込み時に SAS データセットオプションを使用する

次の例では、ハッシュオブジェクトを宣言するときにさまざまな SAS データセットオプションを使用します。

> 重要　ハッシュオブジェクトが CAS で処理される場合、SAS データセットオプションはサポートされません。

```
data x;
  retain j 999;
  do i = 1 to 20;
    output;
  end;
run;

/* Using the WHERE option. */
data _null_;
  length i 8;
  dcl hash h(dataset: 'x (where =(i > 10))', ordered: 'a');
  h.definekey('i');
  h.definedone();
  h.output(dataset: 'example1');
run;
proc print data=example1; run;

/* Using the DROP option. */
data _null_;
  length i 8;
  dcl hash h(dataset: 'x (drop = j)', ordered: 'a');
  h.definekey(all: 'y');
  h.definedone();
  h.output(dataset: 'example2 (where =( i < 8))');
```

```
run;
proc print data=example2; run;

/* Using the FIRSTOBS option. */
data _null_;
  length i j 8;
  dcl hash h(dataset: 'x (firstobs=5)', ordered: 'a');
  h.definekey(all: 'y');
  h.definedone();
  h.output(dataset: 'example3');
run;
proc print data=example3; run;

/* Using the OBS option. */
data _null_;
  length i j 8;
  dcl hash h(dataset: 'x (obs=5)', ordered: 'd');
  h.definekey(all: 'y');
  h.definedone();
  h.output(dataset: 'example4 (rename =(j=k))');
run;
proc print data=example4; run;
```

SAS データセットオプションのリストについては、*SAS データセットオプション: リファレンス*を参照してください。

例 5: 出力データセットにキー集計を追加する

次の例では、変数 *key_sum* は各キーのキー集計持っていて、その変数を出力データセットに追加することを宣言します。

```
data numbers;
  input key_id amount;
  datalines;
    1 10
    2 11
    3 20
    5 5
    4 6
run;

data _null_;
  dcl hash h(dataset:'numbers', suminc: 'i', keysum: 'key_sum');
  h.definekey('key_id');
  h.definedata('key_id', 'amount');
  h.definedone();
  /* avoid uninitialized variable notes */
  call missing(key_id, amount, key_sum);
  i = 1;
  do key_id = 1 to 5;
    rc = h.find();
  end;
  do key_id = 1 to 3;
    rc = h.find();
  end;
  rc = h.output(dataset:'out');
run;
```

```
proc print data=out; run;
```

アウトプット 3.1　キー集計データの出力

Obs	key_id	amount	key_sum
1	2	11	2
2	5	5	1
3	1	10	2
4	3	20	2
5	4	6	1

関連項目

- “コンポーネントオブジェクト”

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

DEFINEDATA メソッド

ハッシュオブジェクトに保存するデータを、指定のデータ変数に関連付けて定義します。

適用対象:　ハッシュオブジェクト

構文

rc=*object*.**DEFINEDATA** ('*datavarname-1*' <, ...'*datavarname-n*'>);

rc=*object*.**DEFINEDATA** (ALL: 'YES'" | YES");

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

'*datavarname*'
データ変数の名前を指定します。

データ変数名は二重引用符で囲むこともできます。

ALL:'YES' | “YES”
すべてのデータ変数をデータに指定して、データセットをオブジェクトコンストラクターに読み込みます。

DECLARE ステートメントまたは_NEW_演算子で *dataset* 引数タグを使用してデータセットを自動的に読み込む場合、ALL: 'YES'を使ってすべてのキー変数を定義できます。

詳細

ハッシュオブジェクトは、ルックアップキーに基づくデータの保存と取得によって機能します。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。ハッシュオブジェクトの初期化を完了するには、すべてのキー変数とデータ変数を定義してから、DEFINEDONE メソッドを呼び出す必要があります。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

注: ADD メソッドまたは REPLACE メソッドにショートカット表記(例: **h.add(key:99, data:'apple', data:'orange')**)を使い、DEFINEDATA メソッドに ALL:'YES' オプションを使う場合は、データの指定順序をデータセット内と同じにする必要があります。

注: ハッシュオブジェクトは値をキー変数(例:**h.find(key:'abc')** >)に割り当てないため、ハッシュオブジェクトとハッシュ反復子が実行するキーとデータの変数割り当てを、SAS コンパイラからは検出できません。したがって、キー変数またはデータ変数への割り当てがプログラムに出現しない場合、変数が初期化されていないことを示す NOTE が発行されます。このような NOTE が発行されないようにするには、次のアクションのいずれかを実行します。

- NONOTES システムオプションを設定します。
- 各キー変数およびデータ変数について、初期割り当てステートメントを(通常は欠損値に)指定します。
- CALL MISSING ルーチンを、すべてのキー変数とデータ変数をパラメーターとして使用します。次に、例を示します。

```
length d $20;
length k $20;
if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k');
  rc = h.defineData('d');
```

```
    rc = h.defineDone();
    call missing(k,d);
  end;
```

DEFINEDATA メソッドの使用方法の詳細については、“キーとデータの定義”を参照してください。

例

次の例では、ハッシュオブジェクトを作成し、キー変数とデータ変数を定義します。

```
data _null_;
  length d $20;
  length k $20;
  /* Declare the hash object and key and data variables */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
run;
```

関連項目

- “キーとデータの定義”

メソッド:

- “DEFINEDONE メソッド” (44 ページ)
- “DEFINEKEY メソッド” (46 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

DEFINEDONE メソッド

キーとデータの定義がすべて完了したことを指定します。

適用対象: ハッシュオブジェクト

構文

rc = *object*.**DEFINEDONE**( );

rc = *object*.**DEFINEDONE** (MEMRC: 'y');

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

memrc:'y'
ハッシュオブジェクトへのデータセットの読み込みでメモリエラーが発生した場合に、エラーからの回復を可能にします。

データセット読み込み時のメモリ不足が原因で呼び出しが失敗すると、ゼロ以外のリターンコードが返されます。ハッシュオブジェクトは基礎配列の主メモリを解放します。この種のエラーが発生した後に実行可能な操作は、DELETE メソッドを使った削除のみです。

詳細

DEFINEDONE メソッド呼び出しでコンストラクターに *dataset* 引数タグを使用すると、データセットはハッシュオブジェクトに読み込まれます。

ハッシュオブジェクトは、ルックアップキーに基づくデータの保存と取得によって機能します。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。ハッシュオブジェクトの初期化を完了するには、すべてのキー変数とデータ変数を定義してから、DEFINEDONE メソッドを呼び出す必要があります。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

DEFINEDONE メソッドの使用方法の詳細については、“キーとデータの定義”を参照してください。

関連項目

- “キーとデータの定義”

メソッド:

- “DEFINEDATA メソッド” (42 ページ)
- “DEFINEKEY メソッド” (46 ページ)

DEFINEKEY メソッド

ハッシュオブジェクトのキー変数を定義します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**DEFINEKEY**('*keyvarname-1* '<, ... '*keyvarname-n*'>);

rc=*object*.**DEFINEKEY**(ALL: 'YES'" | YES");

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

'keyvarname'
キー変数の名前を指定します。

キー変数名は二重引用符で囲むこともできます。

ALL:'YES' | ”YES”
すべてのデータ変数をキーに指定して、データセットをオブジェクトコンストラクターに読み込みます。

DECLARE ステートメントまたは_NEW_演算子で *dataset* 引数タグを使用してデータセットを自動的に読み込む場合、ALL: 'YES'オプションを使ってすべてのキー変数を定義できます。

詳細

ハッシュオブジェクトは、ルックアップキーに基づくデータの保存と取得によって機能します。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を

DEFINEDATA メソッドに渡します。ハッシュオブジェクトの初期化を完了するには、すべてのキー変数とデータ変数を定義してから、DEFINEDONE メソッドを呼び出す必要があります。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

DEFINEKEY メソッドの使用方法の詳細については、“キーとデータの定義”を参照してください。

注: ADD、CHECK、FIND、REMOVE、REPLACE などのメソッドにショートカット表記(例: **h.add(key:99, data:'apple', data:'orange')**)を使い、DEFINEKEY メソッドに ALL:'YES'オプションを使う場合は、キーとデータの指定順序をデータセット内と同じにする必要があります。

注: ハッシュオブジェクトは値をキー変数(例:**h.find(key:'abc')** >)に割り当てないため、ハッシュオブジェクトとハッシュ反復子が実行するキーとデータの変数割り当てを、SAS コンパイラからは検出できません。したがって、キー変数またはデータ変数への割り当てがプログラムに出現しない場合、変数が初期化されていないことを示す NOTE が発行されます。このような NOTE が発行されないようにするには、次のアクションのいずれかを実行します。

- NONOTES システムオプションを設定します。
- 各キー変数およびデータ変数について、初期割り当てステートメントを(通常は欠損値に)指定します。
- CALL MISSING ルーチンを、すべてのキー変数とデータ変数をパラメーターとして使用します。次に、例を示します。

```
length d $20;
length k $20;
if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k');
  rc = h.defineData('d');
  rc = h.defineDone();
  call missing(k, d);
end;
```

関連項目

- “キーとデータの定義”

メソッド:

- “DEFINEDATA メソッド” (42 ページ)
- “DEFINEDONE メソッド” (44 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

DELETE メソッド: ハッシュオブジェクトとハッシュ反復子オブジェクト

ハッシュオブジェクトまたはハッシュ反復子オブジェクトを削除します。

適用対象: ハッシュオブジェクト, ハッシュ反復子オブジェクト

構文

rc=*object*.**DELETE**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

object
ハッシュオブジェクトまたはハッシュ反復子オブジェクトの名前を指定します。

詳細

DATA ステップのコンポーネントオブジェクトは、DATA ステップの終了時に自動的に削除されます。他のハッシュオブジェクトコンストラクターまたはハッシュ反復子オブジェクトでオブジェクト参照名を再利用する場合、またはそのオブジェクトで使用されるメモリをリリースする場合は、DELETE メソッドでハッシュオブジェクトまたはハッシュ反復子オブジェクトを削除する必要があります。

削除したハッシュオブジェクトまたはハッシュ反復子オブジェクトを使用しようとすると、エラーがログに書き込まれます。

ハッシュオブジェクト内からすべてのアイテムを 削除し、ハッシュオブジェクトを再利用できるように保存するには、“CLEAR メソッド” (30 ページ)を使用します。

DO_OVER メソッド

ハッシュオブジェクトの重複キーのリスト内を移動します。

適用対象: ハッシュオブジェクト

構文

object.**DO_OVER** (<KEY:*keyvalue*>);

引数

object
ハッシュオブジェクト名を指定します。

KEY:*keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

詳細

ハッシュオブジェクトが単一のキーに対して複数の値を持つ場合、DO_OVER メソッドを DO ループの反復内で使用して、重複キー間を移動します。DO_OVER メソッドは最初のメソッド呼び出しでキーを読み込み、キーが最後に到達するまで重複キーリストを移動しつづけます。

注: 反復の途中でキーを切り替えた場合、RESET_DUP メソッドを使用してポインターをリストの最初にリセットしなければなりません。そうしないと、SAS は最初のキーを使い続けます。

例

次の例では、重複キーを含むデータセット **dup** を作成します。DO_OVER メソッドと RESET_DUP メソッドを使って、重複キー間を移動します。

```
data dup;
  input key_id value;
  datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
```

```
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
run;

data _null_;
  dcl hash h(dataset:'dup', multidata: 'y', ordered: 'y');
  h.definekey('key_id');
  h.definedata('key_id', 'value');
  h.definedone();
  call missing(key_id, value);

  h.reset_dup();
  key_id = 2;
  do while(h.do_over(key:key_id) eq 0);
    put key_id= value=;
  end;

  key_id = 3;
  do while(h.do_over(key:key_id) eq 0);
    put key_id= value=;
  end;

  key_id = 2;
  do while(h.do_over(key:key_id) eq 0);
    put key_id= value=;
  end;
run;
```

次の行が SAS ログに書き出されます。

```
key_id=2 value=11
key_id=2 value=16
key_id=2 value=9
key_id=3 value=20
key_id=3 value=100
key_id=2 value=11
key_id=2 value=16
key_id=2 value=9
```

関連項目

メソッド:

- “RESET_DUP メソッド” (96 ページ)

EQUALS メソッド

2 つのハッシュオブジェクトが等しいかどうかを確認します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**EQUALS** (HASH: '*object*', RESULT: *variable _name*);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

HASH:'*object*'
1 つ目のハッシュオブジェクトに対して比較する 2 つ目のハッシュオブジェクトの名前を指定します。

RESULT: *variable_name*
結果を保持する数値変数を指定します。ハッシュオブジェクトが等しい場合、結果変数は 1 です。それ以外の場合の結果変数はゼロです。

詳細

次の例では、H1 と H2 のハッシュオブジェクトを比較しています。

```
data compare;
  declare hash h1();
  h1.defineKey('k');
  h1.defineDone();

  declare hash h2();
  h2.defineKey('k');
  h2.defineDone();
  call missing(k);

  rc = h1.equals(hash: 'h2', result: eq);
  if eq then
    put 'hash objects are equal';
  else
```

```
      put 'hash objects are not equal';
run;
```

次の行が SAS ログに書き込まれます。

```
ハッシュオブジェクトが等しい
```

これらの条件がすべて揃ったとき、2 つのハッシュオブジェクトは等しいと定義されます。

- 2 つのハッシュオブジェクトが同サイズである(HASHEXP サイズが等しい)。
- 2 つのハッシュオブジェクトに同数のアイテムがある(H1.NUM_ITEMS = H2.NUM_ITEMS)。
- 2 つのハッシュオブジェクトのキー構造とデータ構造が同一である。
- H1 と H2 ハッシュオブジェクトの比較を順不同で反復した場合に、H1 からの一連のレコードが対応する H2 のレコードと同じキーおよびデータのフィールドを持っている。すなわち、各ハッシュオブジェクトの同じ位置に各レコードがあり、それらのレコードが対応するもう一方のハッシュオブジェクトのレコードと同一である。

例: 2 つのハッシュオブジェクトの比較

次の例では、H1 と H2 のハッシュオブジェクトが宣言され、インスタンス化されています。キーの値が 99 のレコードが両方のオブジェクトに追加されます。EQUALS の最初の呼び出しは、ハッシュオブジェクトが等価であることを示す 0 以外の結果値を返します。

そして、H2 のキーを 100 という値に置き換えます。2 回目の EQUALS の呼び出しでは、H1 と H2 のハッシュオブジェクトを比較し、ハッシュオブジェクトが等価でなくなったことを示す 0 の結果値を返します。

```
data _null_;
  declare hash h1();
  h1.defineKey('k');
  h1.defineDone();

  declare hash h2();
  h2.defineKey('k');
  h2.defineDone();

  k = 99;
  h1.add();
  h2.add();
  rc = h1.equals(hash: 'h2', result: eq);
  put eq=;

put rc=;
  k = 100;
  h2.replace();

  rc = h1.equals(hash: 'h2', result: eq);
  put eq=;
```

```
put rc=;
 run;
```

次の行が SAS ログに書き出されます。

```
eq=1
eq=0
```

FIND メソッド

指定されたキーがハッシュオブジェクトに格納されているかどうかを判断し、見つかった場合はデータセットのデータ変数を更新します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**FIND** (<KEY: *keyvalue-1*, ...KEY: *keyvalue-n*>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

“KEY: *keyvalue*”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで FIND メソッドを使用して、ハッシュオブジェクト内のデータを検索できます。

1 つ目は、このコードに示すように、キーを指定してから FIND メソッドを使用する方法です。

```
data _null_;
```

```
  length k $8  d $12;
  /* Declare hash object and key and data variables */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
  /* Define constant key and data values */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  /* Find the key JOYCE */
  k = 'Joyce';
  rc = h.find();
  if (rc = 0) then
    put 'Key is in the hash object.';
run;
```

2 つ目は、このコードに示すように、ショートカットを使用して、FIND メソッド呼び出しでキーを直接指定する方法です。

```
data _null_;
  length k $8  d $12;
  /* Declare hash object and key and data variables */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
   /* Define constant key and data values */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
 /* Find the key JOYCE */
  rc = h.find(key: 'Joyce');
  if (rc = 0) then
    put 'Key is in the hash object.';
run;
```

ハッシュオブジェクトの各キーに複数のデータアイテムがある場合は、“FIND_NEXT メソッド” (55 ページ)と “FIND_PREV メソッド” (57 ページ)を FIND メソッドと組み合わせ、複数データアイテムのリスト内を移動します。

比較

FIND メソッドは、キーがハッシュオブジェクト内に格納されているかどうかを示す値を返します。FIND メソッドでは、キーがハッシュオブジェクトに存在する場合、同時にデータアイテムの値がデータ変数に設定されます。これによって、メソッド呼び出し後に値が使えるようになります。CHECK メソッドは、キーがハッシュオブジェクトに格納されているかどうかを示す値のみを返します。データ変数は更新されません。

例: FIND メソッドを使ってハッシュオブジェクト内のキーを検索する

次の例では、ハッシュオブジェクトを作成します。2 つのデータ値を追加します。FIND メソッドを使用してハッシュオブジェクト内のキーを検索します。データ値は、キーに関連付けられたデータセット変数に返されます。

```
data _null_;
  length k $8;
  length d $12;
  /* Declare hash object and key and data variable names */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
  /* Define constant key and data values and add to hash object */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  rc = h.add(key: 'Homer', data: 'Odyssey');
  /* Verify that key JOYCE is in hash object and */
  /* return its data value to the data set variable D */
  rc = h.find(key: 'Joyce');
  put d=;
run;
```

d=Ulysses が SAS ログに書き込まれます。

関連項目

- “ハッシュオブジェクトの使用”

メソッド:

- “CHECK メソッド” (28 ページ)
- “DEFINEKEY メソッド” (46 ページ)
- “FIND_NEXT メソッド” (55 ページ)
- “FIND_PREV メソッド” (57 ページ)
- “REF メソッド” (81 ページ)

FIND_NEXT メソッド

現在のキーの複数アイテムリストで現在のリストアイテムを次のアイテムに設定し、対応するデータ変数にデータを設定します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**FIND_NEXT**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

object
ハッシュオブジェクト名を指定します。

詳細

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_NEXT メソッドは、キーに複数のデータアイテムが関連付けられているかどうかを判断します。キーに別のデータアイテムがあることが判明した場合、FIND_NEXT メソッドを使用してそのデータアイテムを 取得できます。これによりデータ変数にデータアイテムの値が設定され、メソッド呼び出し後に使用できるようになります。データアイテムリスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドを使用することでリスト内を移動できます。

例

この例では、FIND_NEXT メソッドを使用して、複数のキーに複数のデータアイテムがあるデータセットを反復処理します。キーに複数のデータアイテムがある 場合、後続のアイテムは **dup** とマークされます。

```
data dup;
  length key_id value 8;
  input key_id value;
  datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
```

```
  5 99
  ;
data _null_;
  dcl hash h(dataset:'dup', multidata: 'y');
  h.definekey('key_id');
  h.definedata('key_id', 'value');
  h.definedone();
  /* avoid uninitialized variable notes */
  call missing (key_id, value);
  do key_id = 1 to 5;
    rc = h.find();
    if (rc = 0) then do;
      put @5 key_id= @15 value=;
      rc = h.find_next();
      do while(rc = 0);
        put 'dup ' @5 key_id= @15 value=;
        rc = h.find_next();
      end;
    end;
  end;
 run;
```

次の行が SAS ログに書き出されます。

```
   key_id=1  value=10
dup key_id=1  value=15
dup key_id=1  value=5
   key_id=2  value=11
dup key_id=2  value=16
dup key_id=2  value=9
   key_id=3  value=20
dup key_id=3  value=100
   key_id=4  value=6
   key_id=5  value=5
dup key_id=5  value=99
```

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “FIND メソッド” (53 ページ)
- “FIND_PREV メソッド” (57 ページ)
- “HAS_NEXT メソッド” (61 ページ)

FIND_PREV メソッド

現在のキーの複数アイテムリストで現在のリストアイテムを次のアイテムに設定し、対応するデータ変数にデータを設定します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**FIND_PREV**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

object
ハッシュオブジェクト名を指定します。

詳細

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_PREV メソッドは、キーに複数のデータアイテムが関連付けられているかどうかを判断します。キーに前のデータアイテムがあることが判明した場合、FIND_PREV メソッドを使用してそのデータアイテムを 取得できます。これによりデータ変数にデータアイテムの値が設定され、メソッド呼び出し後に使用できるようになります。データアイテムリスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドに加えて HAS_PREV メソッドと FIND_PREV メソッドを使用することで、リスト内を移動できます。 例については、“HAS_NEXT メソッド” (61 ページ)を参照してください。

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “FIND メソッド” (53 ページ)
- “FIND_NEXT メソッド” (55 ページ)
- “HAS_PREV メソッド” (63 ページ)

FIRST メソッド

基となるハッシュオブジェクトの開始値を返します。

適用対象: ハッシュ反復子オブジェクト

構文

rc=*object*.**FIRST**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しにリターンコード変数を指定せずにメソッドが失敗した場合、キーが見つからないことを示すエラーメッセージがログに書き込まれます。ハッシュ反復子は、キーを使ってハッシュテーブルを移動します。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

FIRST メソッドは、ハッシュオブジェクト内の最初のデータアイテムを返します。ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは_NEW_演算子で **ordered: 'yes'**または **ordered: 'ascending'**引数タグを使用した場合、ハッシュオブジェクト内のデータアイテムはキー値の昇順で並べ替えられるため、'最小'キー(最小の数値またはアルファベット順の最初)のデータアイテムが返されます。NEXT メソッドへの繰り返される呼び出しは、ハッシュオブジェクト内を反復して移動し、キーの昇順でデータアイテムを返します。反対に、ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは_NEW_演算子で **ordered: 'descending'**引数タグを使用した場合、ハッシュオブジェクト内のデータアイテムはキー値の降順で並べ替えられるため、'最大'キー(最大の数値またはアルファベット順の最後)のデータアイテムが返されます。NEXT メソッドへの繰り返される呼び出しは、ハッシュオブジェクト内を反復して移動し、キーの降順でデータアイテムを返します。

ハッシュオブジェクト内の最後のデータアイテムを返すには、LAST メソッドを使用します。

注: FIRST メソッドは、データ変数にデータアイテムの値を設定し、メソッド呼び出し後にそのデータアイテムを使用できるようにします。

例: ハッシュオブジェクトデータの取得

次の例では、売上データを含むデータセットを作成します。製品を売上順に表示します。データがハッシュオブジェクト内に読み込まれ、データの取得に FIRST メソッドと NEXT メソッドが使用されます。

```
data work.sales;
  input prod $1-6 qty $9-14;
  datalines;
banana  398487
apple   384223
orange  329559
;
data _null_;
    length prod $10 qty $6;
  /* Declare hash object and read SALES data set as ordered */
  if _N_ = 1 then do;
    declare hash h(dataset: 'work.sales', ordered: 'yes');
    declare hiter iter('h');
    /* Define key and data variables */
    h.defineKey('qty');
    h.defineData('prod', 'qty');
    h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(qty, prod);
  end;
  /* Iterate through the hash object and output data values */
  rc = iter.first();
  do while (rc = 0);
    put prod=  qty=;
    rc = iter.next();
  end;
run;
```

次の行が SAS ログに書き出されます。

```
prod=orange qty=329559
prod=apple qty=384223
prod=banana qty=398487
```

関連項目

- 2 章, “ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用”

メソッド:

- “LAST メソッド” (65 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

HAS_NEXT メソッド

現在のキーの複数データアイテムリスト内に次のアイテムがあるかどうかを判断します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**HAS_NEXT** (RESULT: R);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

RESULT:R
数値変数 **R** を指定します。この変数は、データアイテムリスト内に別のデータアイテムがない場合はゼロ値、データアイテムリスト内に別のデータアイテムがある場合はゼロ以外の値を受け取ります。

詳細

1 つのキーに複数のデータアイテムがある場合、HAS_NEXT メソッドを使用して、現在のキーの複数データアイテムリスト内に次のアイテムがあるかどうかを判断できます。別のアイテムがある場合、このメソッドは数値変数 **R** でゼロ以外の値を返します。別のアイテムがない場合、ゼロを返します。

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_NEXT メソッドは、キーに複数のデータアイテムが関連付けられているかどうかを判断します。キーに別のデータアイテムがあることが判明した場合、FIND_NEXT メソッドを使用してそのデータアイテムを取得できます。これによりデータ変数にデータアイテムの値が設定され、メソッド呼び出し後に使用できるようになります。データアイテムリスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドに加えて HAS_PREV メソッドと FIND_PREV メソッドを使用することで、リスト内を移動できます。

例: データアイテムの検索

この例では、複数のキーに複数のデータアイテムがあるハッシュオブジェクトを作成します。さらに、HAS_NEXT メソッドを使用してすべてのデータアイテムを 検索します。

```
data billing;
  length customer_num invoice_amount  8;
  input customer_num invoice_amount;
  datalines;
  1 100
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
data _null_;
  length r 8;
  dcl hash h(dataset:'billing', multidata: 'y');
  h.definekey('customer_num');
  h.definedata('customer_num', 'invoice_amount');
  h.definedone();
  call missing (customer_num, invoice_amount);
  do customer_num = 1 to 5;
    rc = h.find();
    if (rc = 0) then do;
      put @5 customer_num= @20 invoice_amount=;
      h.has_next(result: r);
      do while(r ne 0);
         rc = h.find_next();
         put 'dup ' @5 customer_num= @20 invoice_amount=;
         h.has_next(result: r);
      end;
    end;
  end;
run;
```

次の行が SAS ログに書き出されます。

```
   customer_num=1 invoice_amount=100
dup customer_num=1 invoice_amount=15
dup customer_num=1 invoice_amount=5
   customer_num=2 invoice_amount=11
dup customer_num=2 invoice_amount=16
dup customer_num=2 invoice_amount=9
   customer_num=3 invoice_amount=20
dup customer_num=3 invoice_amount=100
   customer_num=4 invoice_amount=6
   customer_num=5 invoice_amount=5
dup customer_num=5 invoice_amount=99
```

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “FIND メソッド” (53 ページ)
- “FIND_NEXT メソッド” (55 ページ)
- “FIND_PREV メソッド” (57 ページ)
- “HAS_PREV メソッド” (63 ページ)

HAS_PREV メソッド

現在のキーの複数データアイテムリスト内に前のアイテムがあるかどうかを判断します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**HAS_PREV** (RESULT: R);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

RESULT:R
数値変数 **R** を指定します。この変数は、データアイテムリスト内に別のデータアイテムがない場合はゼロ値、データアイテムリスト内に別のデータアイテムがある場合はゼロ以外の値を受け取ります。

詳細

1 つのキーに複数のデータアイテムがある場合、HAS_PREV メソッドを使用して、現在のキーの複数データアイテムリスト内に前のアイテムがあるかどうかを判断できます。前のアイテムがある場合、このメソッドは数値変数 **R** でゼロ以外の値を返します。別のアイテムがない場合、ゼロを返します。

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_NEXT メソッドは、キーに複数のデータアイテムが関連付けられているかどうかを判断します。キーに前のデータアイテムがあることが判明した場合、FIND_PREV メソッドを使用してそのデータアイテムを 取得できます。これによりデータ変数にデータアイテムの値が設定され、メソッド呼び出し後に使用できるようになります。データアイテムリスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドに加えて HAS_PREV メソッドと FIND_PREV メソッドを使用することで、リスト内を移動できます。 例については、“HAS_NEXT メソッド” (61 ページ)を参照してください。

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “FIND メソッド” (53 ページ)
- “FIND_NEXT メソッド” (55 ページ)
- “FIND_PREV メソッド” (57 ページ)
- “HAS_NEXT メソッド” (61 ページ)

ITEM_SIZE 属性

ハッシュオブジェクト内のアイテムのサイズ(バイト)を返します。

適用対象: ハッシュオブジェクト

構文

variable_name=*object*.**ITEM_SIZE**;

引数

variable_name
ハッシュオブジェクト内のアイテムのサイズを 含む変数名を指定します。

object
ハッシュオブジェクト名を指定します。

詳細

ITEM_SIZE 属性は、アイテムのサイズ(バイト)を返します。これには、キーおよびデータ変数の他に、いくつかの追加内部情報が含まれます。ハッシュオブジェクトが ITEM_SIZE 属性と NUM_ITEMS 属性で使用するメモリ量の推定値を取得できます。ITEM_SIZE 属性は、ハッシュオブジェクトが必要とする初期オーバーヘッドを反映せず、必要な内部配置も考慮しません。そのため、ITEM_SIZE では正確なメモリ使用量は提供されず、近似値が返されます。

例: ハッシュアイテムのサイズを返す

次の例では、ITEM_SIZE を使用して MYHASH 内のアイテムのサイズを返します。

```
data work.stock;
  input prod $1-10 qty 12-14;
  datalines;
broccoli 345
corn 389
potato 993
onion 730
;
data _null_;
  if _N_ = 1 then do;
    length prod $10;
  /* Declare hash object and read STOCK data set as ordered */
    declare hash myhash(dataset: "work.stock");
    /* Define key and data variables */
    myhash.defineKey('prod');
    myhash.defineData('qty');
    myhash.defineDone();
  end;
  /* Add a key and data value to the hash object */
  prod = 'celery';
  qty = 183;
  rc = myhash.add();

  /* Use ITEM_SIZE to return the size of the item in hash object */
  itemsize = myhash.item_size;
  put itemsize=;
run;
```

itemsize=40 が SAS ログに書き込まれます。

LAST メソッド

基となるハッシュオブジェクトの最終値を返します。

適用対象: ハッシュ反復子オブジェクト

構文

rc=*object*.**LAST**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しにリターンコード変数を指定せずにメソッドが失敗した場合、キーが見つからないことを示すエラーメッセージがログに書き込まれます。ハッシュ反復子は、キーを使ってハッシュテーブルを 移動します。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

LAST メソッドは、ハッシュオブジェクト内の最後のデータアイテムを返します。ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは _NEW_演算子で **ordered: 'yes'**または **ordered: 'ascending'**引数タグを使用した場合、ハッシュオブジェクト内のデータアイテムはキー値の昇順で並べ替えられるため、'最大'キー(最大の数値またはアルファベット順の最後)のデータアイテムが返されます。反対に、ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは_NEW_演算子で **ordered: 'descending'**引数タグを使用した場合、ハッシュオブジェクト内のデータアイテムはキー値の降順で並べ替えられるため、'最小'キー(最小の数値またはアルファベット順の最初)のデータアイテムが返されます。

ハッシュオブジェクト内の最初のデータアイテムを返すには、FIRST メソッドを使用します。

注: LAST メソッドは、データ変数にデータアイテムの値を設定し、メソッド呼び出し後にそのデータアイテムを使用できるようにします。

関連項目

- 2 章, “ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用”

メソッド:

- “FIRST メソッド” (58 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

NEW 演算子: ハッシュとハッシュ反復子

ハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンスを作成します。

適用対象: ハッシュオブジェクト, ハッシュ反復子オブジェクト

構文

object-reference = **_NEW_** *object* (<*argument_tag-1*: *value-1* <, ...*argument_tag-n*: *value-n*>>);

引数

object-reference
ハッシュオブジェクトまたはハッシュ反復子オブジェクトのオブジェクト参照名を指定します。

object
コンポーネントオブジェクトを指定します。次のいずれかのオブジェクトを指定できます。

hash ハッシュオブジェクトを示します。ハッシュオブジェクトは、迅速なデータの保存および取得のメカニズムを提供します。ハッシュオブジェクトは、ルックアップキーに基づいてデータを保存および取得します。

hiter ハッシュ反復子オブジェクトを示します。ハッシュ反復子オブジェクトを使用すると、ハッシュオブジェクトのデータを正または逆のキー順序で取得できます。

参照項目: 2 章, “ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用”

argument_tag:*value*
ハッシュオブジェクトのインスタンスの作成に使用する情報を指定します。

有効なハッシュオブジェクトの引数タグおよび値は、次のとおりです。

dataset: '*dataset_name* <(*datasetoption*)>'
ハッシュオブジェクトに読み込む SAS データセットの名前を指定します。

SAS データセットの名前には、リテラルまたは文字変数を使用できます。データセット名は単一引用符または二重引用符で囲む必要があります。マクロ変数は二重引用符で囲む必要があります。

DATASET 引数タグでハッシュオブジェクトを宣言するときに、SAS データセットオプションを使用できます。データセットオプションは、それらが表示される SAS データセットにのみ適用されるアクションを指定し、次の操作を実行できるようにします。

- 変数名を変更する
- 処理するオブザベーション番号に基づくオブザベーションのサブセットの選択
- WHERE オプションを使用したオブザベーションの選択
- ハッシュオブジェクトに読み込まれたデータセット、または OUTPUT メソッド呼び出しで指定されている出力データセットの変数の削除または保持
- データセットのパスワードを指定する

次の構文が使用されます。

```
dcl hash h;
h = _new_ hash (dataset: 'x (where = (i > 10))');
```

SAS データセットオプションのリストについては、*SAS データセットオプション: リファレンス*を参照してください。

注 データセットに重複するキーが含まれている場合、デフォルトでは、ハッシュオブジェクトの最初のインスタンスが保持され、後続のインスタンスは無視されます。重複するキーがある場合に、ハッシュオブジェクトの最後のインスタンスを保存するか、または SAS ログにエラーメッセージを書き込むには、DUPLICATE 引数タグを使用します。

duplicate: '*option*'

ハッシュオブジェクトにデータセットを読み込むときに重複するキーを無視するかどうかを判断します。デフォルトでは、最初のキーが保存され、後続の重複はすべて無視されます。オプションには、次のいずれかの値を使用できます。

'replace'
'r'
最後の重複するキーレコードを保存します。

'error'
'e'
重複するキーが検出された場合に、ログにエラーを報告します。

REPLACE オプションを使用する次の例では、キー 531 には **blue**、キー 620 には **brown** が格納されます。デフォルトを使用する場合は、531 には **yellow**、620 には **green** が格納されます。

```
data table;
 input color_id color_name $;
 datalines;
 531 yellow
 620 green
 531 blue
 908 orange
 620 brown
 143 purple
```

```
 run;
data _null_;
length color_id 8 color_name $ 8;
if (_n_ = 1) then do;
   declare hash myhash;
   myhash = _new_ hash (dataset: "table", duplicate: "r");
   rc = myhash.definekey('color_id');
   rc = myhash.definedata('color_id', 'color_name');
   myhash.definedone();
 end;
rc = myhash.output(dataset:"otable");
run;
```

hashexp: *n*

ハッシュオブジェクトの内部テーブルサイズです。ハッシュテーブルのサイズは 2^n です。

HASHEXP の値は、ハッシュテーブルサイズを作成する 2 の累乗指数として使用されます。たとえば、HASHEXP の値 4 は、ハッシュテーブルサイズ 2^4(つまり 16)と同じです。HASHEXP の最大値は 20 です。

ハッシュテーブルサイズは、保存可能なアイテム数とは等しくありません。ハッシュテーブルを、一連の'バケット'と考えます。ハッシュテーブルサイズ 16 には、16 個の'バケット'があります。各バケットに保持できるアイテムの数は無限です。ハッシュテーブルの効率は、アイテムをバケットにマップし、バケットからアイテムを取得するハッシュ関数の機能にあります。

ハッシュオブジェクトのルックアップルーチンの効率を最大にするには、ハッシュオブジェクトのデータ量に応じてハッシュテーブルサイズを設定する必要があります。最適な結果が得られるまで、さまざまな HASHEXP 値を試してみてください。たとえば、ハッシュオブジェクトに 100 万個のアイテムが含まれている場合、ハッシュテーブルサイズ 16(HASHEXP = 4)でも機能しますが、効率は良くありません。ハッシュテーブルサイズ 512 または 1024(HASHEXP = 9 または 10)を使用すると、最高の処理速度が得られます。

デフォルト　8。これは、ハッシュテーブルサイズ 2^8(つまり 256)と同じです。

keysum:'*variable-name*'

すべてのキーのキー集計をトラッキングする変数の名前を指定します。キー集計はキーが何回 FIND メソッド呼び出しに参照されたかのカウントです。

注　キー集計は出力データセットにあります。

ordered: '*option*'

ハッシュオブジェクトとハッシュ反復子オブジェクトを併用する場合、またはハッシュオブジェクト OUTPUT メソッドを使用する場合、データがキー値の順序で返されるかどうか、またはどのように返されるかを指定します。

引数値は二重引用符で囲むこともできます。

option には、次のいずれかの値を使用できます。

'ascending' | 'a'	データはキー値の昇順で返されます。'**ascending**'の指定は、'**yes**'を指定することと同じです。
'descending' | 'd'	データはキー値の降順で返されます。

'YES' | 'Y'　　データはキー値の昇順で返されます。'**yes**'の指定は、'**ascending**'を指定することと同じです。

'NO' | 'N'　　データは未定義の順序で返されます。

デフォルト　NO

multidata: '*option*'
各キーに複数のデータアイテムを許可するかどうかを指定します。

引数値は二重引用符で囲むこともできます。

option には、次のいずれかの値を使用できます。

'YES' | 'Y'　各キーに複数のデータアイテムが許可されます。

'NO' | 'N'　各キーにデータアイテムが 1 つのみ許可されます。

デフォルト　NO

参照項目:　"キー値とデータのペアの重複"

suminc: '*variable-name*'
ハッシュオブジェクトキーの集計数を維持します。SUMINC 引数タグは、DATA ステップ変数を指定します。この変数には、合計増分が保持されます。合計増分はキーが参照されるたびにキー集計に追加される数です。例えば、キー集計は、DATA ステップ変数の現在の値を使用して変更されます。

```
dcl hash myhash(suminc: 'count');
```

詳細については、"キー集計の維持"を参照してください。

参照項目:　"コンストラクターを使ったハッシュオブジェクトデータの初期化" および"ハッシュ反復子オブジェクトの宣言とインスタンス化"

詳細

SAS プログラムで DATA ステップコンポーネントオブジェクトを使用するには、オブジェクトを宣言して作成(インスタンス化)する必要があります。DATA ステップコンポーネントインターフェイスは、DATA ステップ内から定義済みのコンポーネントオブジェクトにアクセスするメカニズムを提供します。

_NEW_演算子を使用してコンポーネントオブジェクトをインスタンス化する場合は、最初に DECLARE ステートメントを使用してコンポーネントオブジェクトを宣言する必要があります。たとえば、次のコード行では、DECLARE ステートメントは、オブジェクト参照 H がハッシュオブジェクトであることを SAS に示します。_NEW_演算子でハッシュオブジェクトが作成され、オブジェクト参照 H に割り当てられます。

```
declare hash h();
h = _new_ hash( );
```

注: DECLARE ステートメントを使用してハッシュオブジェクトまたはハッシュ反復子オブジェクトの宣言とインスタンス化を 1 つのステップで実行できます。

コンストラクターは、コンポーネントオブジェクトをインスタンス化し、コンポーネントオブジェクトデータを初期化するために使用するメソッドです。たとえば、次のコード行では、_NEW_演算子でハッシュオブジェクトがインスタンス化され、オブジェクト参照 H に割り当てられます。さらに、データセット WORK.KENNEL がハッシュオブジェクトに読み込まれます。

```
declare hash h();
h = _new_ hash(datset: "work.kennel");
```

定義済みの DATA ステップコンポーネントオブジェクトおよびコンストラクターの詳細については、“コンストラクターを使ったハッシュオブジェクトデータの初期化”を参照してください。

比較

ハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンスを宣言し、インスタンス化するには、DECLARE ステートメントと_NEW_演算子を使用するか、DECLARE ステートメントのみを使用します。

例: _NEW_演算子を使用したハッシュオブジェクトデータのインスタンス化と初期化

この例では、_NEW_演算子を使用して、ハッシュオブジェクトのデータをインスタンス化して初期化し、ハッシュ反復子オブジェクトをインスタンス化します。

ハッシュオブジェクトにはデータが含まれており、データをキー順序で取得するために反復子が使用されます。

```
data kennel;
  input name $1-10 kenno $14-15;
  datalines;
Charlie     15
Tanner      07
Jake        04
Murphy      01
Pepe        09
Jacques     11
Princess Z  12
;
run;
data _null_;
  if _N_ = 1 then do;
    length kenno $2;
    length name $10;
    /* Declare the hash object */
    declare hash h();
    /* Instantiate and initialize the hash object */
```

```
    h = _new_ hash(dataset:"work.kennel", ordered: 'yes');
    /* Declare the hash iterator object */
    declare hiter iter;
    /* Instantiate the hash iterator object */
    iter = _new_ hiter('h');
    /* Define key and data variables */
    h.defineKey('kenno');
    h.defineData('name', 'kenno');
    h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(kenno, name);
  end;
  /* Find the first key in the ordered hash object and output to the log */
  rc = iter.first();
  do while (rc = 0);
    put kenno '   ' name;
    rc = iter.next();
  end;
run;
```

次の行が SAS ログに書き出されます。

```
NOTE: There were 7 observations read from the data set WORK.KENNEL.
01   Murphy
04   Jake
07   Tanner
09   Pepe
11   Jacques
12   Princess Z
15   Charlie
```

関連項目

- “コンポーネントオブジェクト”

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

NEXT メソッド

基となるハッシュオブジェクトの次の値を返します。

適用対象: ハッシュ反復子オブジェクト

構文

rc=*object*.**NEXT**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しにリターンコード変数を指定せずにメソッドが失敗した場合、キーが見つからないことを示すエラーメッセージがログに書き込まれます。ハッシュイテレータは、キーを使ってハッシュテーブルを移動します。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

NEXT メソッドを繰り返し使用することで、ハッシュオブジェクト内を移動し、データアイテムをキーの順序で返すことができます。

FIRST メソッドは、ハッシュオブジェクト内の最初のデータアイテムを返します。

ハッシュオブジェクト内の前のデータアイテムを返すには、PREV メソッドを使用できます。

注: NEXT メソッドは、データ変数にデータアイテムの値を設定し、メソッド呼び出し後にそのデータアイテムを使用できるようにします。

注: FIRST メソッドを呼び出さずに NEXT メソッドを呼び出した場合でも、NEXT メソッドはハッシュオブジェクト内の最初のアイテムから開始します。

関連項目

- “ハッシュオブジェクトの使用”

メソッド:

- “FIRST メソッド” (58 ページ)
- “PREV メソッド” (80 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

■ “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

NUM_ITEMS 属性

ハッシュオブジェクト内のアイテム数を返します。

適用対象: ハッシュオブジェクト

構文

variable_name=*object*.**NUM_ITEMS**;

引数

variable_name
ハッシュオブジェクト内のアイテム数を含む変数名を指定します。

object
ハッシュオブジェクト名を指定します。

例: ハッシュオブジェクト内のアイテム数を返す

この例では、データセットを作成し、そのデータセットをハッシュオブジェクト内に読み込みます。アイテムがハッシュオブジェクトに追加され、その結果のハッシュオブジェクト内のアイテムの合計数が NUM_ITEMS 属性によって返されます。

```
data work.stock;
  input item $ qty;
  datalines;
broccoli 345
corn 389
potato 993
onion 730
;
data _null_;
  if _N_ = 1 then do;
    length item $10;
    length qty 8;
    length totalitems 8;
    /* Declare hash object and read STOCK data set as ordered */
    declare hash myhash(dataset: "work.stock");
    /* Define key and data variables */
    myhash.defineKey('item');
    myhash.defineData('qty');
    myhash.defineDone();
  end;
```

```
  /* Add a key and data value to the hash object */
  item = 'celery';
  qty = 183;
  rc = myhash.add();
  if (rc ne 0) then
    put 'Add failed';
  /* Use NUM_ITEMS to return updated number of items in hash object */
  totalitems = myhash.num_items;
  put totalitems=;
run;
```

totalitems=5 が SAS ログに書き込まれます。

OUTPUT メソッド

それぞれハッシュオブジェクトにデータが格納された 1 つ以上のデータセットを作成します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**OUTPUT**(< DATASET: '*dataset-1* <(*datasetoption-1 datasetoption-2*...)> ' , ...DATASET: '*dataset-n* <(*datasetoption-1 datasetoption-2*...)>' >);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

DATASET: '*dataset*'
出力データセット名を指定します。

SAS データセットの名前には、文字リテラルまたは文字変数を使用できます。データセット名は二重引用符で囲むこともできます。出力データセットの名前を指定するときには、DATASET 引数タグで SAS データセットオプションを使用できます。マクロ変数は二重引用符で囲む必要があります。

datasetoption
データセットオプションを指定します。

データセットオプションを指定する方法の詳細については、“構文” (*SAS データセットオプション: リファレンス*)を参照してください。

詳細

ハッシュオブジェクトキーは、出力データセットの一部として自動的には保存されません。キーは、DEFINEDATA メソッドを使って、出力データセットに含まれるデータアイテムとして定義できます。また、データアイテムが DEFINEDATA メソッドを使用して定義されていない場合、キーが OUTPUT メソッドで指定されたデータセットに書き込まれます。

ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは_NEW_演算子で **ordered: 'yes'**または **ordered: 'ascending'**引数タグを使用する場合、データアイテムはキー値の昇順でデータセットに書き込まれます。ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントまたは_NEW_演算子で **ordered: 'descending'**引数タグを使用する場合、データアイテムはキー値の降順でデータセットに書き込まれます。**ordered** 引数タグを使用しない場合、順序は未定義になります。

出力データセットの名前を指定するときには、DATASET 引数タグで SAS データセットオプションを使用できます。データセットオプションは、それらが表示される SAS データセットにのみ適用されるアクションを指定し、次の操作を実行できるようにします。

- 変数名を変更する
- 処理するオブザベーション番号に基づくオブザベーションのサブセットの選択
- WHERE オプションを使用したオブザベーションの選択
- ハッシュオブジェクトに読み込まれたデータセット、または OUTPUT メソッド呼び出しで指定されている出力データセットの変数の削除または保持

  注: 削除または保持する変数は、DEFINEDATA メソッドまたは DEFINEKEY メソッドを使用することにより、ハッシュ テーブルに含めておく必要があります。そうでない場合、エラーが発生します。

- データセットのパスワードを指定する

この例では、WHERE データセットオプションを使用して、OUT という出力データセットの特定のデータを選択します。

```
data x;
 do i = 1 to 20;
   output;
 end;
run;

/* Using the WHERE option. */
data _null_;
  length i 8;
  dcl hash h(dataset:'x');
  h.definekey(all: 'y');
  h.definedone();
  h.output(dataset: 'out (where =( i < 8))');
run;
```

この例では、RENAME データセットオプションを使用して、OUT という出力データセットの変数名を J から K に変更します。

```
data x;
 do i = 1 to 20;
   output;
 end;
run;
/* Using the RENAME option. */
data _null_;
  length i j 8;
  dcl hash h(dataset:'x');
  h.definekey(all: 'y');
  h.definedone();
  h.output(dataset: 'out (rename =(i=k))');
run;
```

データセットオプションのリストについては、*SAS データセットオプション: リファレンス*を参照してください。

注: OUTPUT メソッドを使用してデータセットを作成すると、ハッシュオブジェクトは出力データセットには含まれません。この例では、H2 ハッシュオブジェクトは出力データセットから省略され、警告が SAS ログに書き込まれます。

```
data _null_;
  length k 8;
  length d $10;
  declare hash h2();
  declare hash h(ordered: 'y');
  h.defineKey('k');
  h.defineData('k', 'd', 'h2');
  h.defineDone();
  k = 99;
  d = 'abc';
  h.add();
  k = 199;
  d = 'def';
  h.add();
  h.output(dataset:'work.x');
run;
```

例

次のコードでは、天文学データを含むデータセット ASTRO を使用して、メシエ(OBJ)天体が赤経(RA)値の昇順で並べ替えられたハッシュオブジェクトを作成します。このコードは、OUTPUT メソッドを使用してデータをデータセットに格納しています。

```
data astro;
input obj $1-4 ra $6-12 dec $14-19;
datalines;
 M31 00 42.7 +41 16
 M71 19 53.8 +18 47
 M51 13 29.9 +47 12
 M98 12 13.8 +14 54
 M13 16 41.7 +36 28
 M39 21 32.2 +48 26
```

```
 M81 09 55.6 +69 04
M100 12 22.9 +15 49
 M41 06 46.0 -20 44
 M44 08 40.1 +19 59
 M10 16 57.1 -04 06
 M57 18 53.6 +33 02
  M3 13 42.2 +28 23
 M22 18 36.4 -23 54
 M23 17 56.8 -19 01
 M49 12 29.8 +08 00
 M68 12 39.5 -26 45
 M17 18 20.8 -16 11
 M14 17 37.6 -03 15
 M29 20 23.9 +38 32
 M34 02 42.0 +42 47
 M82 09 55.8 +69 41
 M59 12 42.0 +11 39
 M74 01 36.7 +15 47
 M25 18 31.6 -19 15
;
run;
data _null_;
  if _N_ = 1 then do;
    length obj $10;
    length ra $10;
    length dec $10;
    /* Read ASTRO data set as ordered */
  declare hash h(hashexp: 4, dataset:"work.astro", ordered: 'yes');
    /* Define variables RA and OBJ as key and data for hash object */
    h.defineKey('ra');
    h.defineData('ra', 'obj');
    h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(ra, obj);
end;
/* Create output data set from hash object */
rc = h.output(dataset: 'work.out');
run;

proc print data=work.out;
  var ra obj;
  title 'Messier Objects Sorted by Right-Ascension Values';
run;
```

アウトプット 3.2　赤経値で並べ替えられたメシエ天体

Messier Objects Sorted by Right-Ascension Values

Obs	ra	obj
1	00 42.7	M31
2	01 36.7	M74
3	02 42.0	M34
4	06 46.0	M41
5	08 40.1	M44
6	09 55.6	M81
7	09 55.8	M82
8	12 13.8	M98
9	12 22.9	M100
10	12 29.8	M49
11	12 39.5	M68
12	12 42.0	M59
13	13 29.9	M51
14	13 42.2	M3
15	16 41.7	M13
16	16 57.1	M10
17	17 37.6	M14
18	17 56.8	M23
19	18 20.8	M17
20	18 31.6	M25
21	18 36.4	M22
22	18 53.6	M57
23	19 53.8	M71
24	20 23.9	M29
25	21 32.2	M39

関連項目

- “データの格納と取得”

メソッド:

- “DEFINEDATA メソッド” (42 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

PREV メソッド

基となるハッシュオブジェクトの前の値を返します。

適用対象: ハッシュ反復子オブジェクト

構文

rc=*object*.**PREV**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しにリターンコード変数を指定せずにメソッドが失敗した場合、キーが見つからないことを示すエラーメッセージがログに書き込まれます。ハッシュイテレータは、キーを使ってハッシュテーブルを移動します。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

ハッシュオブジェクトを移動して逆のキー順序でデータアイテムを返すには、PREV メソッドを反復して使用します。

FIRST メソッドは、ハッシュオブジェクト内の最初のデータアイテムを返します。LAST メソッドは、ハッシュオブジェクト内の最後のデータアイテムを返します。

ハッシュオブジェクトの次のデータアイテムを返すには、NEXT メソッドを使用できます。

注: PREV メソッドは、データ変数にデータアイテムの値を設定し、メソッド呼び出し後にそのデータアイテムを使用できるようにします。

関連項目

- 2 章, “ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用”

メソッド:

- “FIRST メソッド” (58 ページ)
- “LAST メソッド” (65 ページ)
- “NEXT メソッド” (72 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

REF メソッド

CHECK メソッドと ADD メソッドを 1 つのメソッド呼び出しに統合します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**REF** (<<KEY: *keyvalue-1*>, ...<KEY: *keyvalue-n*>, <DATA: *datavalue-1*>, ...<DATA: *datavalue-n*>>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 *keyvalue* にコンポーネントオブジェクトを含めることはできません。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

"DATA: *datavalue*"ペアの数は、DEFINEDATA メソッドを使用して定義されたデータ変数の数によって変わります。

制限事項 *datavalue* にコンポーネントオブジェクトを含めることはできません。

詳細

CHECK メソッドと ADD メソッドを 1 つの REF メソッド呼び出しに統合できます。次のコードを変更できます。

```
rc = h.check();
 if (rc ne 0) then
   rc = h.add();
```

to

```
rc = h.ref();
```

REF メソッドは、ハッシュオブジェクト内の各キーの出現数を数えるのに便利です。REF メソッドは、最初の ADD で各キーのキー集計を初期化し、後続の CHECK が出現するたびに ADD を変更します。

キー集計の詳細については、“キー集計の維持”を参照してください。

例: REF メソッドを使用したキー集計

次の例では、キー集計に REF メソッドを使用します。

```
data keys;
 input key_id;
datalines;
1
2
1
```

```
3
5
2
3
2
4
1
5
1
;
data key_count;
 length count key_id 8;
 keep count key_id;
 if _n_ = 1 then do;
  declare hash myhash(suminc: "count", ordered: "y");
  declare hiter iter("myhash");
  myhash.defineKey('key_id');
  myhash.defineDone();
  count = 1;
 end;
 do while (not done);
  set keys end=done;
  rc = myhash.ref();
 end;
 rc = iter.first();
 do while(rc = 0);
  rc = myhash.sum(sum: count);
  output;
  rc = iter.next();
 end;
 stop;
run;

proc print data=key_count;
run;
```

アウトプット 3.3　REF メソッドを使用した COUNT の出力

Obs	count	key_id
1	4	1
2	3	2
3	2	3
4	1	4
5	2	5

関連項目

メソッド:

- “ADD メソッド” (26 ページ)
- “CHECK メソッド” (28 ページ)

REMOVE メソッド

指定したキーに関連付けられているデータをハッシュオブジェクトから削除します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**REMOVE** (<KEY: *keyvalue-1*, ...KEY: *keyvalue-n*>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定されている対応するキー変数と型が一致するキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 関連付けられているハッシュ反復子が *keyvalue* を示している場合、REMOVE メソッドはハッシュオブジェクトからキーまたはデータを削除しません。エラーメッセージが発行されます。

詳細

REMOVE メソッドは、ハッシュオブジェクトからキーとデータの両方を削除します。

次の 2 つの方法のいずれかで REMOVE メソッドを使用して、ハッシュオブジェクトのキーおよびデータを削除できます。

1 つ目は、このコードに示すように、キーを指定してから REMOVE メソッドを使用する方法です。

```
data _null_;
  length k $8;
```

```
  length d $12;
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  /* Specify the key */
  k = 'Joyce';
  /* Use the REMOVE method to remove the key and data */
  rc = h.remove();
  if (rc = 0) then
    put 'Key and data removed from the hash object.';
run;
```

2 つ目は、このコードに示すように、ショートカットを使用して、REMOVE メソッド呼び出しでキーを直接指定する方法です。

```
data _null_;
  length k $8;
  length d $12;
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  rc = h.add(key: 'Homer', data: 'Iliad');
  /* Specify the key in the REMOVE method parameter */
  rc = h.remove(key: 'Homer');
  if (rc  =0) then
    put 'Key and data removed from the hash object.';
run;
```

注: REMOVE メソッドではデータ変数の値は変更されません。ハッシュオブジェクト内の値のみが削除されます。

注: ハッシュオブジェクトコンストラクターで **multidata:'y'**を指定すると、REMOVE メソッドは指定したキーのすべてのデータアイテムを 削除します。

例: ハッシュテーブルのキーの削除

この例では、ハッシュテーブルのキーを削除する方法を示します。

```
/* Generate test data */
data x;
```

```
  do k = 65 to 70;
    d = byte (k);
    output;
  end;
run;
data _null_;
  length k 8 d $1;
  /* define the hash table and iterator */
  declare hash H (dataset:'x', ordered:'a');
  H.defineKey  ('k');
  H.defineData ('k', 'd');
  H.defineDone ();
  call missing (k,d);
  declare hiter HI ('H');
/*Use this logic to remove a key in the hash object when an*/
/*iterator is pointing to that key. The NEXT method will*/
/*start at the first item in the hash object if it is called*/
/*without calling the FIRST method. */
   do while (hi.next() = 0);
    if flag then rc=h.remove(key:key);
      if d = 'C' then do;
        key=k;
        flag=1;
      end;
      else flag=0;
    end;
 if flag then rc=h.remove(key:key);
 rc = h.output(dataset: 'work.out');
 stop;
run;
proc print;
run;
```

次の出力は、3 番目のオブジェクトのキーとデータ(キー=67、データ=C)が削除されることを示しています。

アウトプット 3.4　出力から削除されるキーとデータ

Obs	k	d
1	65	A
2	66	B
3	68	D
4	69	E
5	70	F

関連項目

- “ハッシュオブジェクトの使用”

メソッド:

- “ADD メソッド” (26 ページ)
- “DEFINEKEY メソッド” (46 ページ)
- “REMOVEDUP メソッド” (87 ページ)

REMOVEDUP メソッド

指定したキーの現在のデータアイテムに関連付けられているデータをハッシュオブジェクトから削除します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**REMOVEDUP** (<KEY: *keyvalue-1*, ...KEY: *keyvalue-n*>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 関連付けられているハッシュ反復子が *keyvalue* を示している場合、REMOVEDUP メソッドはハッシュオブジェクトからキーまたはデータを削除しません。エラーメッセージが発行されます。

詳細

REMOVEDUP メソッドは、ハッシュオブジェクトからキーとデータの両方を削除します。

次の 2 つの方法のいずれかで REMOVEDUP メソッドを使用して、ハッシュオブジェクトのキーおよびデータを削除できます。1 つ目は、キーを指定してから、REMOVEDUP メソッドを使用する方法です。2 つ目は、ショートカットを使用して、REMOVEDUP メソッド呼び出しでキーを直接指定する方法です。

注: REMOVEDUP メソッドでは、データ変数の値は変更されません。ハッシュオブジェクトの値のみが削除されます。

注: キーのデータアイテムリストにあるデータアイテムが 1 つのみの場合は、キーとデータがハッシュオブジェクトから削除されます。

比較

REMOVEDUP メソッドでは、指定したキーの現在のデータアイテムに関連付けられているデータがハッシュオブジェクトから削除されます。REMOVE メソッドでは、指定したキーに関連付けられているデータがハッシュオブジェクトから削除されます。

例: キーの重複するアイテムの削除

この例では、複数のキーに複数のデータアイテムがあるハッシュオブジェクトを作成します。キーの 2 番目のデータアイテムが削除されます。

```
data testdup;
  length key_id value 8;
  input key_id value;
  datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
data _null_;
  length r 8;
  dcl hash h(dataset:'testdup', multidata: 'y', ordered: 'y');
```

```
  h.definekey('key_id');
  h.definedata('key_id', 'value');
  h.definedone();
  call missing (key_id, value);
  do key_id = 1 to 5;
    rc = h.find();
    if (rc = 0) then do;
      h.has_next(result: r);
      if (r ne 0) then do;
        h.find_next();
        h.removedup();
      end;
    end;
  end;
  dcl hiter i('h');
  rc = i.first();
  do while (rc = 0);
    put key_id= value=;
    rc = i.next();
  end;
run;
```

次の行が SAS ログに書き出されます。

```
key_id=1 value=10
key_id=1 value=5
key_id=2 value=11
key_id=2 value=9
key_id=3 value=20
key_id=4 value=6
key_id=5 value=5
```

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “REMOVE メソッド” (84 ページ)

REPLACE メソッド

指定したキーに関連付けられたデータを新しいデータに置き換えます。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**REPLACE** (<<KEY: *keyvalue-1*>, ...<KEY: *keyvalue-n*>, <DATA: *datavalue-1*>,
...<DATA: *datavalue-n*>>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

“KEY: **keyvalue**”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 *keyvalue* にコンポーネントオブジェクトを含めることはできません。

要件 KEY:*keyvalue* 引数は、ハッシュオブジェクト変数名が指定されていないため、ハッシュオブジェクトで定義された順序になっていなければなりません。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

“DATA: **datavalue**”ペアの数は、DEFINEDATA メソッドを使用して定義されたデータ変数の数によって変わります。

制限事項 *datavalue* にコンポーネントオブジェクトを含めることはできません。

要件 DATA:*datavalue* 引数は、ハッシュオブジェクト変数名が指定されていないため、ハッシュオブジェクトで定義された順序になっていなければなりません。

詳細

次の 2 つの方法のいずれかで REPLACE メソッドを使用して、ハッシュオブジェクト内のデータを置き換えることができます。

1 つ目は、次のコードに示すように、キーおよびデータアイテムを定義してから REPLACE メソッドを使用してデータを置き換える方法です。この例では、キー 'Collie'の **Place** データが'BestInShow'から'bis'に変更されます。

> ヒント　キーは defineData メソッドでデータアイテムとして 定義され、データセットに書き込まれるようになっています。

```
data work.dogshow;
  length breed $10 place $10;
  input breed place;
datalines;
Terrier    2nd
Beagle     3rd
Rottweiler 1st
Collie     BestInShow
Poodle     2nd
Boxer      3rd
;

proc print data=work.dogshow;
  title 'Dog Show Data Set Before Replace';
run;

data _null_;
  length breed $10 place $10;
  if _N_ = 1 then do;
    declare hash h(dataset: 'work.dogshow');
    rc = h.defineKey('breed');
    rc = h.defineData('breed', 'place');
    rc = h.defineDone();
  end;
  /* Specify the key and new data value */
  breed = 'Collie';
  place = 'bis';
  /* Call the REPLACE method to replace the data value */
  rc = h.replace();
  /* Write the hash table to the data set. */
  rc = h.output(dataset: 'work.dogshow');
run;

proc print data=work.dogshow;
  title 'Dog Show Data Set After Replace';
run;
```

2 つ目は、次のコードに示すように、ショートカットを使用して、REPLACE メソッド呼び出しでキーおよびデータを直接指定する方法です。

注: キーは DEFINEDATA メソッドでデータアイテムとして定義されているので、REPLACE メソッドではキーとデータアイテムの両方を含める必要があります。

```
data work.dogshow2;
  length breed $10 place $10;
  input breed place;
```

```
datalines;
Terrier    2nd
Beagle     3rd
Rottweiler 1st
Collie     BestInShow
Poodle     2nd
Boxer      3rd
;
proc print data=work.dogshow2;
  title 'Second Dog Show Data Set Before Replace';
run;

data _null_;
  length breed $10 place $10;
  if _N_ = 1 then do;
    declare hash h(dataset:'work.dogshow2');
    rc = h.defineKey('breed');
    rc = h.defineData('breed', 'place');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(breed, place);
  end;
  /* Specify the key and new data value in the REPLACE method */
  rc = h.replace(key: 'Collie', data: 'Collie', data: 'bis');
  /* Write the hash table to the data set. */
  rc = h.output(dataset: 'work.dogshow2');
run;

proc print data=work.dogshow2;
  title 'Second Dog Show Data Set After Replace';
run;
```

注: ハッシュオブジェクトの REPLACE メソッドは、各キーごとにデータ 1 つずつアイテムがあるハッシュテーブルと一緒に使用するためのものです(MULTIDATA: 'NO')。一方、REPLACEDUP メソッドは、各キーごとに複数のデータアイテムがあるハッシュテーブルと一緒に使用するためのものです(MULTIDATA: 'YES')。REPLACE メソッドを呼び出し、multidata:'y'オプションを使用してハッシュオブジェクトを宣言した場合、現在のキーに対するすべてのデータアイテムが 、新しいデータに置き換えられます。MULTIDATA オプションの詳細については、“DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)を参照してください。

注: REPLACE メソッドを呼び出してキーが見つからなかった場合、キーとデータがハッシュオブジェクトに追加されます。

注: REPLACE メソッドでは、データ変数の値はデータアイテムの値に置き換えられません。ハッシュオブジェクトの値のみが置き換えられます。

比較

REPLACE メソッドでは、指定したキーに関連付けられたデータが新しいデータに置き換えられます。REPLACEDUP メソッドでは、現在のキーの現在のデータアイテムに関連付けられたデータが新しいデータに置き換えられます。

関連項目

- “ハッシュオブジェクトの使用”

メソッド:

- “DEFINEDATA メソッド” (42 ページ)
- “DEFINEKEY メソッド” (46 ページ)
- “REPLACEDUP メソッド” (93 ページ)

REPLACEDUP メソッド

現在のキーの現在のデータアイテムに関連付けられたデータを新しいデータに置き換えます。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**REPLACEDUP** (<DATA: *datavalue-1*, ...DATA: *datavalue-n*>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

“DATA: *datavalue*”ペアの数は、DEFINEDATA メソッドを使用して現在のキーに定義されたデータ変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで REPLACEUP メソッドを使用して、ハッシュオブジェクト内のデータを置き換えることができます。

1 つ目は、データアイテムを定義してから、REPLACEDUP メソッドを使用する方法です。2 つ目は、ショートカットを使用して、REPLACEDUP メソッド呼び出しでデータを直接指定する方法です。

注: REPLACEDUP メソッドを呼び出してキーが見つからなかった場合、キーとデータがハッシュオブジェクトに追加されます。

注: REPLACEDUP メソッドでは、データ変数の値はデータアイテムの値に置き換えられません。ハッシュオブジェクト内の値のみが置換されます。

比較

REPLACEDUP メソッドでは、現在のキーの現在のデータアイテムに関連付けられたデータが新しいデータに置き換えられます。REPLACE メソッドでは、指定したキーに関連付けられたデータが新しいデータに置き換えられます。

例: 現在のキーのデータの置換

この例では、複数のキーに複数のデータアイテムがあるハッシュオブジェクトを作成します。重複データアイテムが見つかった場合、データアイテムの値に 300 が加算されます。

```
data testdup;
  length key_id value 8;
  input key_id value;
  datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
data _null_;
  length r 8;
  dcl hash h(dataset:'testdup', multidata: 'y', ordered: 'y');
  h.definekey('key_id');
  h.definedata('key_id', 'value');
```

```
  h.definedone();
  call missing (key_id, value);
  do key_id = 1 to 5;
    rc = h.find();
    if (rc = 0) then do;
      put @5 key_id= @15 value=;
      h.has_next(result: r);
      do while(r ne 0);
        rc = h.find_next();
        put 'dup ' @5 key_id= @15 value=;
        value = value + 300;
        rc = h.replacedup();
        h.has_next(result: r);
      end;
    end;
  end;
  put 'iterating...';
  dcl hiter i('h');
  rc = i.first();
  do while (rc = 0);
    put @5 key_id= @15 value=;
    rc = i.next();
   end;
run;
```

次の行が SAS ログに書き出されます。

```
   key_id=1  value=10
dup key_id=1  value=15
dup key_id=1  value=5
   key_id=2  value=11
dup key_id=2  value=16
dup key_id=2  value=9
   key_id=3  value=20
dup key_id=3  value=100
   key_id=4  value=6
   key_id=5  value=5
dup key_id=5  value=99
iterating...
   key_id=1  value=10
   key_id=1  value=315
   key_id=1  value=305
   key_id=2  value=11
   key_id=2  value=316
   key_id=2  value=309
   key_id=3  value=20
   key_id=3  value=400
   key_id=4  value=6
   key_id=5  value=5
   key_id=5  value=399
```

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “REPLACE メソッド” (89 ページ)

RESET_DUP メソッド

DO_OVER メソッド使用時にポインターをキーの重複リストの最初にリセットします。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**RESET_DUP**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

詳細

ハッシュオブジェクトが単一のキーに対して複数の値を持つ場合、DO_OVER メソッドを DO ループの反復内で使用して、重複キー間を移動します。DO_OVER メソッドは最初のメソッド呼び出しでキーを読み込み、キーが最後に到達するまで重複キーリストを移動し続けます。

反復の途中でキーを切り替えた場合、RESET_DUP メソッドを使用してポインターをリストの最初にリセットしなければなりません。そうしないと、SAS は最初のキーを使い続けます。

サンプルは DO_OVER メソッド例 (49 ページ)を参照してください。

関連項目

メソッド:

- “DO_OVER メソッド” (49 ページ)

SETCUR メソッド

反復を開始するキーアイテムを 指定します。

適用対象: ハッシュ反復子オブジェクト

構文

rc=*object*.**SETCUR** (KEY: '*keyvalue-1*' <, ...KEY: '*keyvalue-n*'>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

KEY: '*keyvalue*'
反復の開始キーとしてのキー値を指定します。

詳細

ハッシュ反復子を使うと、ハッシュオブジェクト内のどのアイテムでも反復を開始できます。SETCUR メソッドは、反復の開始キーを設定します。開始アイテムの指定には、KEY オプションを使用します。

例: 開始キーアイテムの 指定

次の例では、天文学データを含むデータセット ASTRO を作成します。データにはメシエ天体(OBJ)と赤経(RA)の値が含まれています。メソッドは、最初のアイテムや最後のアイテムではなく、RA= 18 31.6 から反復を開始します。データがハッシュオブジェクト内に読み込まれ、反復を開始するのに SETCUR メソッドが使用されます。*ordered* 引数タグが YES に設定されているため、出力が昇順で並べ替えられています。

```
data work.astro;
input obj $1-4 ra $6-12 dec $14-19;
datalines;
 M31 00 42.7 +41 16
```

```
 M71 19 53.8 +18 47
 M51 13 29.9 +47 12
 M98 12 13.8 +14 54
 M13 16 41.7 +36 28
 M39 21 32.2 +48 26
 M81 09 55.6 +69 04
M100 12 22.9 +15 49
 M41 06 46.0 -20 44
 M44 08 40.1 +19 59
 M10 16 57.1 -04 06
 M57 18 53.6 +33 02
  M3 13 42.2 +28 23
 M22 18 36.4 -23 54
 M23 17 56.8 -19 01
 M49 12 29.8 +08 00
 M68 12 39.5 -26 45
 M17 18 20.8 -16 11
 M14 17 37.6 -03 15
 M29 20 23.9 +38 32
 M34 02 42.0 +42 47
 M82 09 55.8 +69 41
 M59 12 42.0 +11 39
 M74 01 36.7 +15 47
 M25 18 31.6 -19 15
;
```

次のコードでは、反復の開始キーが**'18 31.6'**に設定されます。

```
data _null_;
length obj $10;
length ra $10;
length dec $10;

declare hash myhash(hashexp: 4, dataset:"work.astro", ordered:"yes");

declare hiter iter('myhash');
myhash.defineKey('ra');
myhash.defineData('obj', 'ra');
myhash.defineDone();
call missing (ra, obj, dec);
rc = iter.setcur(key: '18 31.6');
  do while (rc = 0);
    put obj= ra=;
    rc = iter.next();
  end;
run;
```

次の行が SAS ログに書き出されます。

```
obj=M25 ra=18 31.6
  obj=M22 ra=18 36.4
  obj=M57 ra=18 53.6
  obj=M71 ra=19 53.8
  obj=M29 ra=20 23.9
  obj=M39 ra=21 32.2
```

最初のアイテムまたは最後のアイテムから反復を開始するには、FIRST メソッドまたは LAST メソッドをそれぞれ使用します。

関連項目

- 2 章, “ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用”

メソッド:

- “FIRST メソッド” (58 ページ)
- “LAST メソッド” (65 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

SUM メソッド

ハッシュテーブルから特定のキーの集計値を取得し、その値を DATA ステップ変数に保存します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**SUM** (<KEY: *keyvalue-1*, ...KEY: *keyvalue-n*,> SUM: *variable-name*);

必須引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

“KEY: *keyvalue*”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

SUM: *variable-name*
特定のキーの現在の集計値を保存する DATA ステップ変数を指定します。

詳細

ハッシュオブジェクトからキーの集計値を取得するには、SUM メソッドを使用します。詳細については、“キー集計の維持”を参照してください。

比較

SUM メソッドでは、1 つのキーにつき 1 つのデータアイテムしか存在しないときに、特定のキーの集計値を取得します。SUMDUP メソッドでは、1 つのキーにつき複数のデータアイテムが存在するときに、現在のキーの現在のデータアイテムについて集計値を取得します。

例: 特定のキーのキー集計値の取得

次の例では、2 つのキー K=99 および K=100 を指定し、SUM メソッドを使用して各キーの集計値を取得します。

```
data _NULL_;
  retain total 0;
  if _N_=1 then
    do;
      declare hash h(suminc:'count');
      rc=h.defineKey('k');
      rc=h.definedone();
    end;
  k = 99;
  count = 1;
  h.add();
  /* key=99 summary is now 1 */
  k = 100;
  h.add();
  /* key=100 summary is now 1 */
  k = 99;
  h.find();
  /* key=99 summary is now 2 */
  count = 2;
  h.find();
  /* key=99 summary is now 4 */
  k = 100;
  h.find();
  /* key=100 summary is now 3 */
  h.sum(sum: total);
  put 'total for key 100 = ' total;
  k = 99;
  h.sum(sum:total);
  put 'total for key 99 = ' total;
run;
```

最初の PUT ステートメントは k=100 の集計値を出力します。

```
total for key 100 = 3
```

2 番目の PUT ステートメントは k=99 の集計値を出力します。

```
total for key 99 = 4
```

関連項目

メソッド:

- “ADD メソッド” (26 ページ)
- “FIND メソッド” (53 ページ)
- “CHECK メソッド” (28 ページ)
- “DEFINEKEY メソッド” (46 ページ)
- “REF メソッド” (81 ページ)
- “SUMDUP メソッド” (101 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (32 ページ)

SUMDUP メソッド

現在のキーの現在のデータアイテムについて 集計値を取得し、その値を DATA ステップ変数に保存します。

適用対象: ハッシュオブジェクト

構文

rc=*object*.**SUMDUP** (SUM: *variable-name*);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

object
ハッシュオブジェクト名を指定します。

SUM: *variable-name*
現在のキーの現在のデータアイテムについて取得した集計値を保存する DATA ステップ変数を指定します。

詳細

キーに複数のデータアイテムがある場合、ハッシュオブジェクトからキーの集計値を取得するには、SUMDUP メソッドを使用します。詳細については、“キー集計の維持”を参照してください。

比較

SUMDUP メソッドでは、1 つのキーにつき複数のデータアイテムが存在するときに、現在のキーの現在のデータアイテムについて集計値を取得します。SUM メソッドでは、1 つのキーにつき 1 つのデータアイテムしか存在しないときに、特定のキーの集計値を取得します。

例: 集計値の取得

次の例では、SUMDUP メソッドを使って、現在のデータアイテムの集計値を取得します。このメソッドは、HAS_PREV メソッドと FIND_PREV メソッドを使うと、リストの前方にループできることも示しています。FIND_PREV メソッドは、現在のリストアイテムについては FIND_NEXT メソッドと同様の処理を行います。異なるのは、複数のアイテムリストで前方に移動する点です。

```
data dup;
  length key_id value 8;
  input key_id value;
  cards;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
data _null_;
  length r i sum 8;
  i = 0;
  dcl hash h(dataset:'dup', multidata: 'y', suminc: 'i');
```

```
 h.definekey('key_id');
 h.definedata('key_id', 'value');
 h.definedone();
 call missing (key_id, value);
 i = 1;
do key_id = 1 to 5;
  rc = h.find();
  if (rc = 0) then do;
    h.has_next(result: r);
    do while(r ne 0);
      rc = h.find_next();
      rc = h.find_prev();
      rc = h.find_next();
      h.has_next(result: r);
    end;
  end;
end;
i = 0;
do key_id = 1 to 5;
 rc = h.find();
 if (rc = 0) then do;
    h.sum(sum: sum);
    put @5 key_id= @15 value= @26 sum=;
    h.has_next(result: r);
    do while(r ne 0);
      rc = h.find_next();
      h.sumdup(sum: sum);
      put 'dup ' @5 key_id= @15 value= @26 sum=;
      h.has_next(result: r);
    end;
  end;
end;
run;
```

次の行が SAS ログに書き出されます。

```
   key_id=1  value=10   sum=2
dup key_id=1  value=15   sum=3
dup key_id=1  value=5    sum=2
   key_id=2  value=11   sum=2
dup key_id=2  value=16   sum=3
dup key_id=2  value=9    sum=2
   key_id=3  value=20   sum=2
dup key_id=3  value=100  sum=2
   key_id=4  value=6    sum=1
   key_id=5  value=5    sum=2
dup key_id=5  value=99   sum=2
```

このメソッドがどのように機能するかを確認するために、3 つの値 10、15、5 (この順番で格納)を持つ key_id 1 を考えてみましょう。

```
   key_id=1  value=10   sum=2
dup key_id=1  value=15   sum=3
dup key_id=1  value=5    sum=2
```

最初の do key_id = 1 to 5;ループの値リストを移動中、最初の FIND メソッド呼び出しで値アイテム 10 のキー集計値が 1 に設定されます。最初の FIND_NEXT メソッド呼び出しで値アイテム 15 のキー集計値が 1 に設定されます。次の FIND_PREV メ

ソッド呼び出しでは、値アイテム 10 に戻り、キー集計値が 2 に増分されます。さらに、最後の FIND_NEXT メソッド呼び出しで、値アイテム 15 のキー集計値が 2 に設定されます。ループの次の反復では、値アイテム 5 のキー集計値が 1 に設定され、値アイテム 15 のキー集計値が 3 に設定されます。最後に、値アイテム 5 のキー集計値が 2 に増分されます。

この例では、リストに以前のエントリが存在していることがわかっているため、FIND_PREV メソッドを呼び出す前に HAS_PREV メソッドを呼び出していません。存在しなかった場合は、ループに入りません。

このことから、重複処理によっては特殊なメソッドが必要なこともわかります。(この場合、現在の値アイテムのキー集計値を取得することで、SUMDUP メソッドは SUM メソッドと同様の処理を行います。)

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “SUM メソッド” (99 ページ)

4

Java オブジェクトの使用

Java オブジェクトの使用 **105**
Java オブジェクトについて 105
CLASSPATH と Java オプション 105
Java オブジェクト使用に関する制限と要件 106
Java オブジェクトの宣言とインスタンス化 106
オブジェクトフィールドへのアクセス 107
オブジェクトメソッドへのアクセス 108
型の問題 108
Java オブジェクトと配列 110
Java オブジェクト引数を渡す 111
Java 例外 113
Java 標準出力 114
Java オブジェクトの例 114

Java オブジェクトの使用

Java オブジェクトについて

Java クラスをインスタンス化して結果オブジェクトのフィールドとメソッドにアクセスする、Java ネイティブインターフェイス (JNI)に似たメカニズムを提供します。Java と DATA ステップコードの両方を含むハイブリッドアプリケーションを作成できます。

CLASSPATH と Java オプション

SAS のこれまでのバージョンでは、Java クラスの検索に JREOPTIONS システムオプションを使用していました。

Java オブジェクトが Java クラスを検索できるように CLASSPATH 環境変数を設定する必要があります。使用するクラスは、CLASSPATH に含まれる必要があります。クラスが.jar ファイルにある場合、その.jar ファイル名が CLASSPATH に含まれる必要があります。

TKS3_CUSTOM_REGION 環境変数を設定するための構文の例は次のとおりです。

- グローバル SAS 構成ファイル set CLASSPATH ~/HelloWorld.jar
- ローカル EXPORT コマンド export CLASSPATH=~/HelloWorld.jar

Java オブジェクト使用に関する制限と要件

Java オブジェクトの使用時に次の制限と要件が適用されます。

- Java オブジェクトは、SAS から Java メソッドを呼び出すために設計されています。Java オブジェクトは、関数の SAS ライブラリを拡張するためのものではありません。特に大きなデータセットを含む場合は、DATA ステップへの高速なインプロセス拡張としては、PROC FCMP 関数を呼び出すほうが効率的です。Java オブジェクトを使用して大きなデータセットに対してこの種類の処理を実行すると、かなりの時間がかかります。
- SAS は、SAS ソフトウェアのインストール中に明示的に要求された Java ランタイム環境(JRE)のみサポートします。
- SAS は、SAS のインストール中に設定された Java オプションのみサポートします。
- Java オブジェクトで使用する前に、Java アプリケーションが正しく実行されることを確認します。
- Java による SAS ログへのテキスト出力の 1 バイト目でのパーセント文字(%)の使用は、SAS によって予約されています。Java テキスト行の 1 バイト目に%を書く必要がある場合は、そのすぐ後に別のパーセントを追加してエスケープします(%%)。
- SAS がロックダウン状態の時、Java オブジェクトは使用できません。詳細については、“LOCKDOWN” (*SAS プログラマガイド: エッセンシャル*)を参照してください。

Java オブジェクトの宣言とインスタンス化

DECLARE ステートメントを使用して、Java オブジェクトを宣言します。新しい Java オブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化して、Java オブジェクト名を引数タグとして使用します。

```
declare javaobj j;
j = _new_ javaobj("somejavaclass");
```

この例では、DECLARE ステートメントは、コンパイラにオブジェクト参照 J の種類は Java ということを示します。つまり、Java オブジェクトのインスタンスは変数 J

に格納されます。ここでは、オブジェクト参照 J のみ宣言しています。Java 型のコンポーネントオブジェクトを保持する可能性があります。Java オブジェクトの宣言は一度のみ行います。_NEW_演算子は Java オブジェクトのインスタンスを作成し、オブジェクト参照 J に割り当てます。Java クラス名の SOMEJAVACLASS は、コンストラクター引数として渡されます。これは、Java オブジェクトコンストラクターに必要な最初で唯一の引数です。その他の引数はすべて Java クラス自体のコンストラクター引数です。

DECLARE ステートメントと_NEW_演算子を使用して Java オブジェクトの宣言とインスタンス化を行う 2 ステップのプロセスの代替手段として、DECLARE ステートメントをコンストラクターメソッドとして使用して、1 ステップで Java オブジェクトの宣言とインスタンス化を行うことができます。構文は次の通りです。

DECLARE JAVAOBJ*object-name*(“*java-class*”, <*argument-1*, ... *argument–n*>);

詳細については、“DECLARE ステートメント: Java オブジェクト” (132 ページ)と“_NEW_ 演算子: Java オブジェクト” (147 ページ)を参照してください。

オブジェクトフィールドへのアクセス

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、DATA ステップのパブリックフィールドとクラスフィールドにアクセスして変更できます。パブリックフィールドは静的でなく、Java クラスで public として宣言されています。クラスフィールドは静的であり、Java クラスからアクセスされます。

静的でないフィールドまたは静的なフィールドにアクセスするのかによって、オブジェクトフィールドにアクセスするメソッド呼び出しは、次のいずれかの形式になります。

GET*type***FIELD**(“*field-name*”, *value*);

GETSTATIC*type***FIELD**(“*field-name*”, *value*);

静的なフィールドまたは静的でないフィールドにアクセスするのかによって、オブジェクトフィールドを変更するメソッド呼び出しは、次のいずれかの形式になります。

SET*type***FIELD**(“*field-name*”, *value*);

SETSTATIC*type***FIELD**(“*field-name*”, *value*);

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題”を参照してください。*field-name* 引数は、Java フィールドの種類を指定します。*value* は、メソッドによって返されるか、または設定される値を指定します。

詳細と例については、“Java オブジェクトの言語要素のディクショナリ”を参照してください。

オブジェクトメソッドへのアクセス

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、DATA ステップのパブリックメソッドとクラスメソッドにアクセスできます。パブリックメソッドは静的でなく、Java クラスで public として宣言されています。クラスメソッドは静的であり、Java クラスからアクセスされます。

静的でないメソッドまたは静的なメソッドにアクセスするのかによって、Java メソッドにアクセスするメソッド呼び出しは、次のいずれかの形式になります。

object.**CALL***type***METHOD** (“*method-name*”, <*method-argument-1 …, method-argument-n*>,
<*return value*>);

object.**CALLSTATIC***type***METHOD** (“*method-name*”,
<*method-argument-1 …, method-argument-n*>, <*return value*>);

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題”を参照してください。

詳細と例については、“Java オブジェクトの言語要素のディクショナリ”を参照してください。

型の問題

Java 型のセットは、SAS データの型のスーパーセットです。Java のデータ型には、標準数値と標準文字値以外に、BYTE、SHORT、CHAR があります。SAS のデータ型は、数値と文字の 2 種類のみです。

表 1.3 には、Java オブジェクトメソッドの呼び出しの使用時における、Java データ型と SAS データ型のマッピングが示されています。

表 4.1 Java データ型と SAS データ型のマッピング方法

Java のデータ型	SAS のデータ型
BOOLEAN	数値
BYTE	数値
CHAR	数値
DOUBLE	数値
FLOAT	数値

Java のデータ型	SAS のデータ型
INT	数値
LONG	数値
SHORT	数値
STRING	文字 [1]

1 Java STRING データ型は、UTF-8 文字列として SAS 文字データの種類にマッピングされています。

STRING 以外は、Java クラスから DATA ステップにオブジェクトを返すことができません。しかし、オブジェクトを Java メソッドに渡すことはできます。詳細については、“Java オブジェクト引数を渡す”を参照してください。

オブジェクトを返す Java メソッドの中には、オブジェクト値を変換するラッパークラスを作成して処理されるメソッドもあります。次の例では、Java ハッシュテーブルはオブジェクト値を返します。しかし、型の変換を処理する単純な Java ラッパークラスを作成することで、DATA ステップからハッシュテーブルを使用できます。DATA ステップから **dhash** クラスと **shash** クラスにアクセスできます。

```
/* Java code */
import java.util.*;

public class dhash
{
  private Hashtable  table;

  public dhash()
  {
    table = new Hashtable  ();
  }

  public void put(double key, double value)
  {
    table.put(new Double(key), new Double(value));
  }

  public double get(double key)
  {
    Double ret = table.get(new Double(key));
    return ret.doubleValue();
  }
}

import java.util.*;

public class shash
{
  private Hashtable  table;

  public shash()
 {
    table = new Hashtable  ();
```

```
  }

  public void put(double key, String value)
 {
    table.put(new Double(key), value);
 }

  public String get(double key)
  {
    return table.get(new Double(key));
  }
}
```

```
/* DATA step code */
data _null_;
  dcl javaobj sh('shash');
  dcl javaobj dh('dhash');
  length d 8;
  length s $20;

  do i = 1 to 10;
    dh.callvoidmethod('vput', i, i * 2);
  end;

  do i = 1 to 10;
    sh.callvoidmethod('put', i, 'abc' || left(trim(i)));  end;

  do i = 1 to 10;
    dh.calldoublemethod('get', i, d);
    sh.callstringmethod('get', i, s);
    put d= s=;
  end;
run;
```

次の行が SAS ログに書き出されます。

```
d=2 s=abc1
d=4 s=abc2
d=6 s=abc3
d=8 s=abc4
d=10 s=abc5
d=12 s=abc6
d=14 s=abc7
d=16 s=abc8
d=18 s=abc9
d=20 s=abc10
```

Java オブジェクトと配列

DATA ステップ配列を Java オブジェクトに渡せます。

次の例では、配列 **d** と配列 **s** が Java オブジェクト **j** に渡されます。

```
/* Java code
*/
```

```
import java.util.*;
import java.lang.*;
class jtest
{
  public void dbl(double args[])
 {
    for(int i = 0; i < args.length; i++)
      System.out.println(args[i]);
 }

 public void str(String args[])
 {
   for(int i = 0; i < args.length; i++)
     System.out.println(args[i]);
 }
}
```

```
/* DATA Step code */
data _null_;
  dcl javaobj j("jtest");
  array s{3} $20 ("abc", "def", "ghi");
  array d{10} (1:10);
  j.callVoidMethod("dbl", d);
  j.callVoidMethod("str", s);
run;
```

次の行が SAS ログに書き出されます。

```
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
abc
def
ghi
```

1 次元配列パラメーターのみサポートされます。しかし、配列が行順に渡されることを利用して、多次元配列引数を渡すことが可能です。この Java コードでは、次元のインデックスを手動で作成する必要があります。つまり、一次元配列パラメーターを宣言して、部分配列のインデックスを作成します。

Java オブジェクト引数を渡す

Java クラスから DATA ステップにオブジェクトを返すことはできませんが、Java クラスにオブジェクトと文字列を渡すことはできます。

たとえば、**java/util/Vector** とその反復子に次のラッパークラスがあるとします。

```
/* Java code */
```

```
import java.util.*;

class mVector extends Vector
{
  public mVector()
 {
    super();
 }

 public mVector(double d)
 {
    super((int)d);
 }

 public void addElement(String s)
 {
    addElement((Object)s);
 }
}

import java.util.*;
public class mIterator
{
  protected mVector m_v;
  protected Iterator iter;

  public mIterator(mVector v)
  {
     m_v = v;
     iter = v.iterator();
  }

 public boolean hasNext()
 {
    return iter.hasNext();
 }

 public String next()
 {
   String ret = null;
   ret = (String)iter.next();
   return ret;
 }
}
```

これらのラッパークラスは、型の変換の実行に役立ちます(たとえば、**mVector** コンストラクターは DOUBLE 引数を使用します)。**java/util/Vector** のコンストラクターは整数値を使用しますが、DATA ステップには整数型がないため、コンストラクターのオーバーロードが必要です。

次の DATA ステッププログラムはこれらのクラスを使用します。プログラムはベクトルを作成してフィルし、反復子のコンストラクターにベクトルを渡し、ベクトルのすべての値をリストします。ベクトルのフィル後に、反復子を作成する必要がある点に注意してください。反復子は、作成時点のベクトルの変更数のコピーを保持します。この値は、ベクトルの最新の変更数と同期されている必要があります。ベクトルのフィル前に反復子を作成すると、コードに例外が発生します。

```
/* DATA step code */
data _null_;
  length b 8;
  length val $200;
  dcl javaobj v("mVector");

  v.callVoidMethod("addElement", "abc");
  v.callVoidMethod("addElement", "def");
  v.callVoidMethod("addElement", "ghi");
  dcl javaobj iter("mIterator", v);

  iter.callBooleanMethod("hasNext", b);
  do while(b);
    iter.callStringMethod("next", val);
    put val=;
    iter.callBooleanMethod("hasNext", b);
  end;

  m.delete();
  v.delete();
  iter.delete();
run;
```

次の行が SAS ログに書き出されます。

```
val=abc
val=def
val=ghi
```

オブジェクトを渡す際の現在の制限の 1 つとして、指定された署名に基づいて、JNI メソッドルックアップルーチンがフルクラスルックアップを実行しないことがあります。つまり、次のコードに示すように、**mIterator** コンストラクターを変更して **Vector** を使用することはできません。

```
/* Java code */
public mIterator(Vector v)
{
   m_v = v;
   iter = v.iterator();
}
```

mVector が **Vector** のサブクラスであっても、メソッドルックアップルーチンはコンストラクターを検索できません。現時点では、新しいメソッドを追加するか、またはラッパークラスを作成して Java で型を管理することでのみ解決できます。

Java 例外

Java 例外は EXCEPTIONCHECK メソッド、EXCEPTIONCLEAR メソッドおよび EXCEPTIONDESCRIBE メソッドを介して処理されます。

メソッド呼び出し時に例外が発生したかどうかを確認するには、EXCEPTIONCHECK メソッドを使います。例外が発生するメソッドを呼び出す場合は、呼び出し後に例外の有無を確認することをお勧めします。例外が発生した場合は、適切な操作を実行してから、EXCEPTIONCLEAR メソッドを使って例外をクリアします。

EXCEPTIONDESCRIBE メソッドは、例外のデバッグログのオンとオフを切り替えます。例外のデバッグログがオンの場合、例外情報が JVM の標準出力に出力されます。デフォルトでは、JVM の標準出力は SAS ログにリダイレクトされます。例外のデバッグはデフォルトでオフとなっています。

詳細については、“EXCEPTIONCHECK メソッド” (135 ページ)、“EXCEPTIONCLEAR メソッド” (137 ページ)、“EXCEPTIONDESCRIBE メソッド” (139 ページ)を参照してください。

Java 標準出力

デフォルトでは、次のように標準出力に送られる Java のステートメントからの出力は SAS ログに送られます。

```
System.out.println("hello");
```

SAS ログに送られる Java 出力は、DATA ステップの終了時にフラッシュされます。このフラッシュにより、Java 出力は DATA ステップの実行中に生成された出力の後に表示されます。FLUSHJAVAOUTPUT メソッドを使用して出力の同期をとると、実行順に表示されます。

Java オブジェクトの例

例 1: シンプルな Java メソッドの呼び出し

この Java クラスは、3 つの数値を合計する簡単なメソッドを作成します。

```
/* Java code */
class MyClass
{
  double compute(double x, double y, double z)
    {
      return (x + y + z);
    }
}

/* DATA step code */
data _null_;
  dcl javaobj j("MyClass");

  rc = j.callDoubleMethod("compute", 1, 2, 3, r);

  put rc= r=;
run;
```

次の行が SAS ログに書き込まれます。

```
rc=0 rc=6
```

例 2: ユーザーインターフェイスの作成

Java コンポーネントアクセスメカニズムを提供することに加えて、Java オブジェクトを使用して単純な Java ユーザーインターフェイスを作成できます。

この Java クラスは、複数のボタンがある単純なユーザーインターフェイスを作成します。また、このユーザーインターフェイスは、ユーザーによって入力されたボタンの選択の順序を表す値のキューも管理します。

```
/* Java code */
import java.awt.*;
import java.util.*;
import java.awt.event.*;

class colorsUI extends Frame
{
  private Button red;
  private Button blue;
  private Button green;
  private Button quit;
  private Vector list;
  private boolean d;
  private colorsButtonListener cbl;

  public colorsUI()
  {
    d = false;
    list = new Vector();
    cbl = new colorsButtonListener();

    setBackground(Color.lightGray);
    setSize(320,100);
    setTitle("New Frame");
    setVisible(true);
    setLayout(new FlowLayout(FlowLayout.CENTER, 10, 15));
    addWindowListener(new colorsUIListener());

    red = new Button("Red");
    red.setBackground(Color.red);
    red.addActionListener(cbl);

    blue = new Button("Blue");
    blue.setBackground(Color.blue);
    blue.addActionListener(cbl);

    green = new Button("Green");
    green.setBackground(Color.green);
    green.addActionListener(cbl);

    quit = new Button("Quit");
    quit.setBackground(Color.yellow);
    quit.addActionListener(cbl);
```

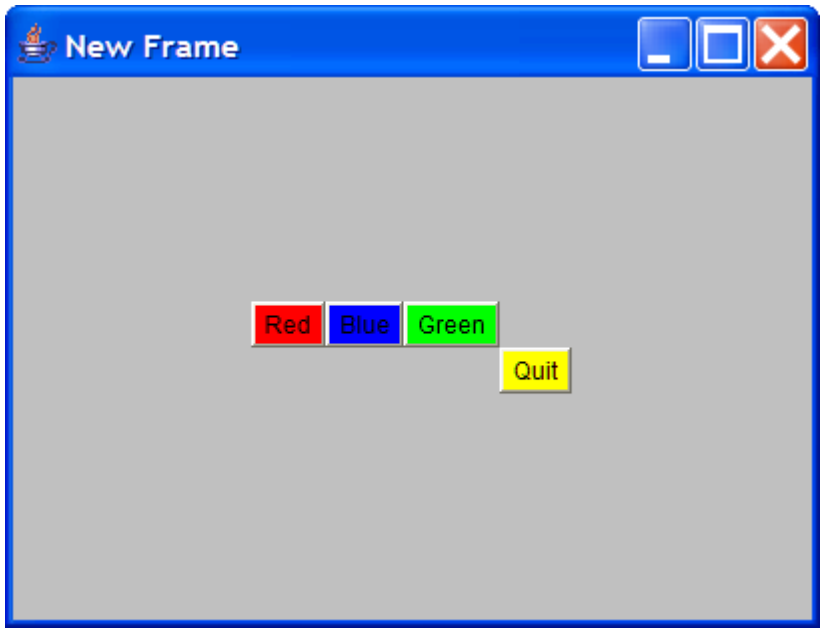
```
  this.add(red);
  this.add(blue);
  this.add(green);
  this.add(quit);

  show();
}

public synchronized void enqueue(Object o)
{
  synchronized(list)
    {
      list.addElement(o);
      notify();
    }
}

public synchronized Object dequeue()
{
  try
    {
      while(list.isEmpty())
        wait();

      if (d)
        return null;

      synchronized(list)
        {
          Object ret = list.elementAt(0);
          list.removeElementAt(0);
          return ret;
        }
    }
  catch(Exception e)
    {
      return null;
    }
}

public String getNext()
{
  return (String)dequeue();
}

public boolean done()
{
  return d;
}

class colorsButtonListener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    Button b;
    String l;
```

```
      b = (Button)e.getSource();
      l = b.getLabel();
      if ( l.equals("Quit") )
      {
        d = true;
        hide();
        l = "";
      }
      enqueue(l);
    }
  }

  class colorsUIListener extends WindowAdapter
  {
    public void windowClosing(WindowEvent e)
    {
      Window w;
      w = e.getWindow();
      d = true;
      enqueue("");
      w.hide();
    }
  }

  public static void main(String s[])
  {
     colorsUI cui;
     cui = new colorsUI();
  }
}
/* DATA step code */
data colors;
  length s $10;
  length done 8;
  drop done;

  if (_n_ = 1) then do;
   /* Declare and instantiate colors object (from colorsUI.class) */
    dcl javaobj j("colorsUI");
  end;

  /*
   * colorsUI.class will display a simple UI and maintain a
   * queue to hold color choices.
   */

  /* Loop until user hits quit button */
  do while (1);
    j.callBooleanMethod("done", done);
    if (done) then
      leave;
    else do;
     /* Get next color back from queue */
      j.callStringMethod("getNext", s);
      if s ne "" then
        output;
```

```
      end;
  end;
run;
proc print data=colors;
run;
quit;
```

DATA ステップコードで、**colorsUI** クラスはインスタンス化され、ユーザーインターフェイスが表示されます。ループを入力します。このループは、**Quit** をクリックすると終了します。このアクションは、Done 変数を介して DATA ステップに伝えられます。ループ中、DATA ステップは Java クラスのキューから値を取得し、出力データセットに値を逐次書き込みます。

図 4.1　Java オブジェクトで作成されるユーザーインターフェイス



例 3: カスタムクラスローダーの作成

クラスパスに Java クラスをすべて 含める必要がない場合があります。独自のクラスローダーを書き、クラスを検索してロードできます。次の例では、カスタムクラスローダーの作成法を示しています。

この例では、クラス **x** を作成し、フォルダーまたはディレクトリ **y** に常駐させています。このクラスのメソッドを、Java オブジェクトと **y** フォルダーを含むクラスパスを使用して呼び出します。

```
/* Java code */
package com.sas;

public class x
{
  public void m()
  {
    System.out.println("method m in y folder");
```

```
  }

  public void m2()
  {
    System.out.println("method m2 in y folder");
  }
}

/* DATA step code */
data _null_;
  dcl javaobj j('com/sas/x');
  j.callvoidmethod('m');
  j.callvoidmethod('m2');
run;
```

次の行が SAS ログに書き出されます。

```
method m in y folder
method m2 in y folder
```

別のクラス **x** が別のフォルダー **z** にあるとします。

```
/* Java code
*/
package com.sas;

public class z
{
  public void m()
  {
    System.out.println("method m in y folder");
  }

  public void m2()
  {
    System.out.println("method m2 in y folder");
  }
}
```

クラスパスを変更することでフォルダー **y** のクラスではなくこのクラスのメソッドを呼び出せますが、SAS の再起動が必要です。次のメソッドは、クラスのロードにより動的なコントロールを可能にします。

カスタムクラスローダーを作成するには、まず、Java オブジェクトから呼び出すメソッドをすべて含むインターフェイスを作成します。このプログラムの場合、**m** と **m2** になります。

```
/* Java
code */
public interface apiInterface
{
  public void m();
  public void m2();
}
```

次に、実際の実装に使用するクラスを作成します。

```
/* Java code */
import com.sas.x;
```

```
public class apiImpl implements apiInterface
{
  private x x;

  public apiImpl()
  {
    x = new x();
  }

  public void m()
  {
    x.m();
  }

  public void m2()
  {
    x.m2();
  }
}
```

Java オブジェクトインスタンスクラスに委任して、これらのメソッドが呼び出されます。**apiClassLoader** カスタムクラスローダーを作成するコードはこのセクションの後半で提供されます。

```
/* Java code */
public class api
{
 /* Load classes from the z folder */
 static ClassLoader customLoader =  new apiClassLoader("C:\\z");
 static String API_IMPL = "apiImpl";
 apiInterface cp = null;

 public api()
  {
    cp = load();
  }

 public void m()
  {
    cp.m();
  }

 public void m2()
  {
    cp.m2();
  }

 private static apiInterface load()
  {
    try
     {
       Class aClass = customLoader.loadClass(API_IMPL);
       return (apiInterface) aClass.newInstance();
     }
    catch (Exception e)
    {
       e.printStackTrace();
```

```
      return null;
    }
  }
}
```

api Java オブジェクトインスタンスクラスを介して委任して、次の DATA ステッププログラムはこれらのメソッドを呼び出します。この Java オブジェクトは、**z** フォルダーからクラスをロードするカスタムクラスローダーを作成する **api** クラスをインスタンス化します。**api** クラスはカスタムローダーを呼び出し、Java オブジェクトに **apiImpl** インターフェイス実装クラスのインスタンスを返します。メソッドが Java オブジェクトから呼び出されると、**api** クラスによってメソッドが実装クラスに委任されます。

```
/* DATA step code */
data _null_;
  dcl javaobj j('api');
  j.callvoidmethod('m');
  j.callvoidmethod('m2');
run;
```

次の行が SAS ログに書き込まれます。

```
method m is z folder
method m2 in z folder
```

これまでの Java コードでは、.jar ファイルを使用して、**ClassLoader** コンストラクターのクラスパスを補うこともできます。

```
static ClassLoader customLoader = new apiClassLoader("C:\\z;C:\\temp\some.jar");
```

この場合、カスタムクラスローダーの Java コードは次のようになります。このクラスローダーのコードは、必要に応じて追加または変更できます。

```
import java.io.*;
import java.util.*;
import java.util.jar.*;
import java.util.zip.*;

public class apiClassLoader extends ClassLoader
{
 //class repository where findClass performs its search
 private List classRepository;

 public apiClassLoader(String loadPath)
  {
    super(apiClassLoader.class.getClassLoader());
    initLoader(loadPath);
  }

 public apiClassLoader(ClassLoader parent,String loadPath)
  {
    super(parent);
    initLoader(loadPath);
  }

 /**
  * This method will look for the class in the class repository. If
  * the method cannot find the class, the method will delegate to its parent
  * class loader.
```

```
 *
 * @param className A String specifying the class to be loaded
 * @return  A Class object loaded by the apiClassLoader
 * @throws ClassNotFoundException if the method is unable to load the class
 */
public Class loadClass(String name) throws ClassNotFoundException
 {
   // Check if the class is already loaded
   Class loadedClass = findLoadedClass(name);

   // Search for class in local repository before delegating
  if (loadedClass == null)
   {
     loadedClass = myFindClass(name);
   }

   // If class not found, delegate to parent
   if (loadedClass == null)
    {
      loadedClass = this.getClass().getClassLoader().loadClass(name);
    }
  return loadedClass;
 }

private Class myFindClass(String className) throws ClassNotFoundException
 {
   byte[] classBytes = loadFromCustomRepository(className);
   if(classBytes != null)
    {
      return defineClass(className,classBytes,0,classBytes.length);
    }
  return null;
 }

/**
 *  This method loads binary class file data from the classRepository.
 */
private byte[] loadFromCustomRepository(String classFileName)
  throws ClassNotFoundException
 {
   Iterator dirs = classRepository.iterator();
   byte[] classBytes = null;
   while (dirs.hasNext())
    {
      String dir = (String) dirs.next();

      if (dir.endsWith(".jar"))
       {
         // Look for class in jar

         String jclassFileName = classFileName;

         jclassFileName = jclassFileName.replace('.', '/');
         jclassFileName += ".class";

         try
```

```
            {
              JarFile j = new JarFile(dir);
              for (Enumeration e = j.entries(); e.hasMoreElements() ;)
                {
                  Object n = e.nextElement();

                  if (jclassFileName.equals(n.toString()))
                   {
                     ZipEntry zipEntry = j.getEntry(jclassFileName);
                     if (zipEntry == null)
                      {
                        return null;
                      }
                     else
                      {
                        // read file
                        InputStream is = j.getInputStream(zipEntry);
                        classBytes = new byte[is.available()];
                        is.read(classBytes);
                        break;
                      }
                   }
                }
              }
             catch (Exception e)
              {
                System.out.println("jar file exception");
                return null;
              }
           }
          else
           {
             // Look for class in directory
             String fclassFileName = classFileName;

             fclassFileName = fclassFileName.replace('.', File.separatorChar);
             fclassFileName += ".class";

             try
              {
                File file = new File(dir,fclassFileName);
                if(file.exists()) {
                  //read file
                  InputStream is = new FileInputStream(file);
                  classBytes = new byte[is.available()];
                  is.read(classBytes);
                  break;
                }
              }
             catch(IOException ex)
              {
                System.out.println("IOException raised while reading class
file data");
                ex.printStackTrace();
                return null;
              }
```

```
            }
          }
        return classBytes;
      }

    private void initLoader(String loadPath)
      {
        /*
         * loadPath is passed in as a string of directories/jar files
         * separated by the File.pathSeparator
         */
        classRepository = new ArrayList();
        if((loadPath != null) && !(loadPath.equals("")))
          {
            StringTokenizer tokenizer =
              new StringTokenizer(loadPath,File.pathSeparator);
            while(tokenizer.hasMoreTokens())
              {
                classRepository.add(tokenizer.nextToken());
              }
          }
      }
  }
```

5

Java オブジェクトの言語要素のディクショナリ

カテゴリ別の Java オブジェクトメソッド **125**

ディクショナリ **127**
CALLtypeMETHOD メソッド 127
CALLSTATICtypeMETHOD メソッド 129
DECLARE ステートメント: Java オブジェクト 132
DELETE メソッド: Java オブジェクト 135
EXCEPTIONCHECK メソッド 135
EXCEPTIONCLEAR メソッド 137
EXCEPTIONDESCRIBE メソッド 139
FLUSHJAVAOUTPUT メソッド 141
GETtypeFIELD メソッド 142
GETSTATICtypeFIELD メソッド 145
NEW 演算子: Java オブジェクト 147
SETtypeFIELD メソッド 149
SETSTATICtypeFIELD メソッド 151

カテゴリ別の Java オブジェクトメソッド

Java オブジェクトメソッドには 5 つのカテゴリがあります。

表 5.1　カテゴリ別の Java オブジェクトメソッド

カテゴリ	説明
削除	Java オブジェクトを削除できます。

カテゴリ	説明
例外	例外情報を収集して例外をクリアできます。
フィールド参照	Java オブジェクトの静的なインスタンスフィールドまたは静的でないインスタンスフィールドの値を返したり設定したりできます。
メソッド参照	静的な Java メソッドと静的でない Java メソッドにアクセスできます。
出力	出力先に Java 出力を即座に送信できます。

次の表に、Java オブジェクトメソッドの概要を示します。詳細については、各メソッドのディクショナリエントリを参照してください。

カテゴリ	**言語要素**	**説明**
削除	DELETE メソッド: Java オブジェクト (p. 135)	Java オブジェクトを削除します。
出力	FLUSHJAVAOUTPUT メソッド (p. 141)	出力先に Java 出力を送信するように指定します。
フィールド参照	GETtypeFIELD メソッド (p. 142)	Java オブジェクトの静的でないフィールドの値を返します。
	GETSTATICtypeFIELD メソッド (p. 145)	Java オブジェクトの静的なフィールドの値を返します。
	SETtypeFIELD メソッド (p. 149)	Java オブジェクトの静的でないフィールドの値を変更します。
	SETSTATICtypeFIELD メソッド (p. 151)	Java オブジェクトの静的フィールドの値を変更します。
メソッド参照	CALLtypeMETHOD メソッド (p. 127)	静的でない Java メソッドから Java オブジェクトのインスタンスメソッドを呼び出します。
	CALLSTATICtypeMETHOD メソッド (p. 129)	静的な Java メソッドから Java オブジェクトのインスタンスメソッドを呼び出します。
例外	EXCEPTIONCHECK メソッド (p. 135)	メソッド呼び出し時に例外が発生したかどうかを確認します。
	EXCEPTIONCLEAR メソッド (p. 137)	現在、発生している例外をクリアします。
	EXCEPTIONDESCRIBE メソッド (p. 139)	デバッグログのオン/オフを切り替え、例外情報を出力します。

ディクショナリ

CALLtypeMETHOD メソッド

静的でない Java メソッドから Java オブジェクトのインスタンスメソッドを呼び出します。

カテゴリ: メソッド参照

適用対象: Java オブジェクト

構文

object.**CALL***type***METHOD** ("*method-name*", <*method-argument-1*, ...*method-argument-n*>,
<*return-value*>);

引数

object
Java オブジェクト名を指定します。

type
静的でない Java メソッドの結果型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
結果型に BOOLEAN を指定します。

BYTE
結果型に BYTE を指定します。

CHAR
結果型に CHAR を指定します。

DOUBLE
結果型に DOUBLE を指定します。

FLOAT
結果型に FLOAT を指定します。

INT
結果型に INT を指定します。

LONG
結果型に LONG を指定します。

SHORT
結果型に SHORT を指定します。

STRING
結果型に STRING を指定します。

VOID
結果型に VOID を指定します。

参照項目: “型の問題”

method-name
静的でない Java メソッドの名前を指定します。

要件 メソッド名は単一引用符または二重引用符で囲む必要があります。

method-argument
メソッドに渡すパラメーターを指定します。

return-value
メソッドで値を返す場合の戻り値を指定します。

詳細

Java オブジェクトをインスタンス化したら、CALL*type*METHOD メソッドを使用した Java オブジェクトでのメソッド呼び出しによって、静的でない Java メソッドにアクセスできます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題”を参照してください。

比較

静的でない Java メソッドには CALL*type*METHOD メソッドを使用します。Java メソッドが静的な場合は、CALLSTATIC*type*METHOD メソッドを使用します。

例: フィールドの値の設定と取得

次の例では、3 つの静的でないフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、CALLC*type*FIELD メソッドを使用してフィールドの値が設定および取得されます。

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttest
{
  public int i;
```

```
  public double d;
  public string s;
  public int im()
  {
    return i;
  }
  public String sm()
  {
     return s;
  }
  public double dm()
  {
    return d;
  }
}

/*  DATA step code */
data _null_;
  dcl javaobj j("ttest");
  length val 8;
  length str $20;
   j.setIntField("i", 100);
  j.setDoubleField("d", 3.14159);
  j.setStringField("s", "abc");
  j.callIntMethod("im", val);
  put val=;
  j.callDoubleMethod("dm", val);
  put val=;
  j.callStringMethod("sm", str);
  put str=;
run;
```

次の行が SAS ログに書き出されます。

```
val=100
val=3.14159
str=abc
```

関連項目

メソッド:

- “CALLSTATICtypeMETHOD メソッド” (129 ページ)

CALLSTATICtypeMETHOD メソッド

静的な Java メソッドから Java オブジェクトのインスタンスメソッドを呼び出します。

カテゴリ: メソッド参照

適用対象: Java オブジェクト

構文

object.**CALLSTATIC***type***METHOD** ("*method-name*", <*method-argument-1* , ...*method-argument-n*>, <*return-value*>);

引数

object
Java オブジェクト名を指定します。

type
静的な Java メソッドの結果型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
結果型に BOOLEAN を指定します。

BYTE
結果型に BYTE を指定します。

CHAR
結果型に CHAR を指定します。

DOUBLE
結果型に DOUBLE を指定します。

FLOAT
結果型に FLOAT を指定します。

INT
結果型に INT を指定します。

LONG
結果型に LONG を指定します。

SHORT
結果型に SHORT を指定します。

STRING
結果型に STRING を指定します。

VOID
結果型に VOID を指定します。

参照項目: “型の問題”

method-name
静的な Java メソッド名を指定します。

要件 メソッド名は単一引用符または二重引用符で囲む必要があります。

method-argument
メソッドに渡すパラメーターを指定します。

return-value
メソッドで値を返す場合の戻り値を指定します。

詳細

Java オブジェクトをインスタンス化したら、CALLSTATIC*type*METHOD メソッドを使用した Java オブジェクトでのメソッド呼び出しによって、静的な Java メソッドにアクセスできます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題”を参照してください。

比較

静的な Java メソッドには CALLSTATIC*type*METHOD メソッドを使用します。Java メソッドが静的でない場合は、CALL*type*METHOD メソッドを使用します。

例: 静的なフィールドの設定と取得

次の例では、3 つの静的なフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、CALLSTATIC*type*FIELD メソッドを使用してフィールドの値が設定および取得されます。

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttestc
{
  public static double d;
  public static double dm()
  {
    return d;
  }
```

```
/* DATA step code */
data x;
  declare javaobj j("ttestc");
  length d 8;
  j.SetStaticDoubleField("d", 3.14159);
  j.callStaticDoubleMethod("dm", d);
  put d=;
run;
```

次の行が SAS ログに書き込まれます。

```
d=3.14159
```

関連項目

メソッド:

- “CALLtypeMETHOD メソッド” (127 ページ)

DECLARE ステートメント: Java オブジェクト

Java オブジェクトを宣言します。Java オブジェクトのインスタンスを作成し、データを初期化します。

別名: DCL

構文

形式 1: **DECLARE JAVAOBJ** *object-reference*;

形式 2: **DECLARE JAVAOBJ** *object-reference* ("*java-class*", <*argument-1*, ... *argument-n*>);

引数

object-reference
Java オブジェクトのオブジェクト参照名を指定します。

java-class
インスタンス化する Java クラスの名前を指定します。

要件 Java クラス名は二重引用符または単一引用符で囲む必要があります。

Java パッケージのパスを指定する場合、パスにはピリオド(.)ではなくフォワードスラッシュ(/)を使用する必要があります。たとえば、*"java.util.Hashtable"*は正しいクラス名ではありません。正しいクラス名は *"java/util/Hashtable"*です。

argument
Java オブジェクトのインスタンスの作成に使用する情報を指定します。
argument の有効な値は Java オブジェクトによって異なります。

参照項目: “DECLARE ステートメントを使用した Java オブジェクトのインスタンス生成(フォーム 2)” (133 ページ)

詳細

基本情報

SAS プログラムで DATA ステップコンポーネントオブジェクトを使用するには、オブジェクトを宣言して作成(インスタンス化)する必要があります。DATA ステップコンポーネントインターフェイスは、DATA ステップ内から定義済みのコンポーネントオブジェクトにアクセスするメカニズムを提供します。

詳細については、“コンポーネントオブジェクト”を参照してください。

Java オブジェクトの宣言(フォーム 1)

DECLARE ステートメントを使用して Java オブジェクトを宣言します。

```
declare javaobj j;
```

DECLARE ステートメントは、オブジェクト参照 J が Java オブジェクトであることを SAS に示します。

新しい Java オブジェクトを宣言した後に、_NEW_演算子を使用してオブジェクトをインスタンス化します。たとえば、次のコード行では、_NEW_演算子で Java オブジェクトが作成され、オブジェクト参照 J に割り当てられます。

```
j = _new_ javaobj("somejavaclass");
```

DECLARE ステートメントを使用した Java オブジェクトのインスタンス生成(フォーム 2)

DECLARE ステートメントと_NEW_演算子を使用して Java オブジェクトを宣言し、インスタンス化する 2 ステップのプロセスの代わりに、DECLARE ステートメントを使用して、Java オブジェクトを 1 つのステップで宣言し、インスタンス化することができます。たとえば、次のコード行では、DECLARE ステートメントで Java オブジェクトが宣言およびインスタンス化され、Java オブジェクトがオブジェクト参照 J に割り当てられます。

```
declare javaobj j("somejavaclass");
```

前述のコード行は、次のコードを使用するのと同等です。

```
declare javaobj j;
j = _new_ javaobj("somejavaclass");
```

constructor は、コンポーネントオブジェクトをインスタンス化し、コンポーネントオブジェクトデータを初期化するために使用できるメソッドです。たとえば、次のコード行では、DECLARE ステートメントで Java オブジェクトが宣言およびインスタンス化され、Java オブジェクトがオブジェクト参照 J に割り当てられます。Java オブジェクトコンストラクターで唯一必要な引数は、インスタンス化する Java クラスの名前です。その他の引数はすべて Java クラス自体のコンストラクター引数です。次の例では、Java クラス名 **testjavaclass** がコンストラクターで、値 **100** と**.8** はコンストラクター引数です。

```
declare javaobj j("testjavaclass", 100, .8);
```

比較

Java オブジェクトを宣言し、インスタンスを作成するには、DECLARE ステートメントと_NEW_演算子を使用するか、DECLARE ステートメントのみを使用します。

例

例 1: DECLARE ステートメントと_NEW_演算子を使用した Java オブジェクトの宣言とインスタンス化

次の例では、単純な Java クラスが作成されます。DECLARE ステートメントと _NEW_演算子を使用して、このクラスのインスタンスを作成します。

```
/* Java code */
import java.util.*;
import java.lang.*;
public class simpleclass
{
  public int i;
  public double d;
}
```

```
/* DATA step code
data _null_;
  declare javaobj myjo;
  myjo = _new_ javaobj("simpleclass");
run;
```

例 2: DECLARE ステートメントを使用した Java オブジェクトの作成とインスタンス化

次の例では、ハッシュテーブルの Java クラスが作成されます。DECLARE ステートメントを使用して、容量および負荷係数を指定し、このクラスのインスタンスを作成します。この例では、DATA ステップの唯一の数値型が Java のデータ型 DOUBLE と同等であるため、ラッパークラス **mhash** が必要です。

```
/* Java code */
import java.util.*;
public class mhash extends Hashtable;
{
  mhash (double size, double load)
    {
      super ((int)size, (float)load);
    }
}
```

```
/* DATA step code */
data _null_;
  declare javaobj h("mhash", 100, .8);
run;
```

関連項目

- “コンポーネントオブジェクト”

演算子:

- “_NEW_ 演算子: Java オブジェクト” (147 ページ)

DELETE メソッド: Java オブジェクト

Java オブジェクトを削除します。

カテゴリ: 削除

適用対象: Java オブジェクト

構文

object.**DELETE**( );

引数

object
Java オブジェクト名を指定します。

詳細

DATA ステップのコンポーネントオブジェクトは、DATA ステップの終了時に自動的に削除されます。他の Java オブジェクトコンストラクターでオブジェクト参照変数を再利用する場合は、DELETE メソッドで Java オブジェクトを削除する必要があります。

削除した Java オブジェクトを使用しようとすると、エラーがログに書き込まれます。

EXCEPTIONCHECK メソッド

メソッド呼び出し時に例外が発生したかどうかを確認します。

カテゴリ: 例外

適用対象: Java オブジェクト

構文

object.**EXCEPTIONCHECK** (*status*);

引数

object
Java オブジェクト名を指定します。

status
返される例外ステータスを指定します。

ヒント Java から返されるステータス値のデータ型は DOUBLE です。これは SAS の数値データ値に対応します。

詳細

Java 例外は EXCEPTIONCHECK メソッド、EXCEPTIONCLEAR メソッドおよび EXCEPTIONDESCRIBE メソッドを介して処理されます。

メソッド呼び出し時に例外が発生したかどうかを確認するには、EXCEPTIONCHECK メソッドを使います。例外が発生する可能性のある Java メソッドを呼び出した後は、毎回 EXCEPTIONCHECK メソッドを呼び出すのが理想的です。

例: 例外の確認

次の例では、例外が発生するメソッドが Java クラスに含まれています。DATA ステップはそのメソッドを呼び出し、例外の有無を確認します。

```
/* Java code */
public class a
  {
    public void m() throws NullPointerException
    {
      throw new NullPointerException();
    }
  }

/* DATA step code */
data _null_;
  length e 8;
  dcl javaobj j('a');
  rc = j.callvoidmethod('m');
  /* Check for exception.  Value is returned in variable 'e' */
  rc = j.exceptioncheck(e);
  if (e) then
    put 'exception';
  else
    put 'no exception';
  run;
```

次の行が SAS ログに書き込まれます。

```
exception
```

関連項目

メソッド:

- “EXCEPTIONCLEAR メソッド” (137 ページ)
- “EXCEPTIONDESCRIBE メソッド” (139 ページ)

EXCEPTIONCLEAR メソッド

現在、発生している例外をクリアします。

カテゴリ: 例外

適用対象: Java オブジェクト

構文

object.**EXCEPTIONCLEAR**( );

引数

object
Java オブジェクト名を指定します。

詳細

Java 例外は EXCEPTIONCHECK メソッド、EXCEPTIONCLEAR メソッドおよび EXCEPTIONDESCRIBE メソッドを介して処理されます。

例外が発生するメソッドを呼び出す場合は、呼び出し後に例外の有無を確認することをお勧めします。例外が発生した場合は、適切な操作を実行してから、EXCEPTIONCLEAR メソッドを使って例外をクリアします。

例外が発生していない場合、このメソッドには何の効果もありません。

例

例 1: 例外の確認とクリア

次の例では、例外が発生するメソッドが Java クラスに含まれています。DATA ステップでこのメソッドを呼び出した後、例外をクリアします。

```
/* Java code */
public class a
```

```
  {
    public void m() throws NullPointerException
    {
      throw new NullPointerException();
    }
  }

/* DATA step code */
data _null_;
  length e 8;
  dcl javaobj j('a');
  rc = j.callvoidmethod('m');
  /* Check for exception.  Value is returned in variable 'e' */
  rc = j.exceptioncheck(e);
  if (e) then
    put 'exception';
  else
    put 'no exception';
  /* Clear the exception and check it again */
  rc = j.exceptionclear( );
  rc = j.exceptioncheck(e);
  if (e) then
    put 'exception';
  else
    put 'no exception';
run;
```

次の行が SAS ログに書き出されます。

```
exception
no exception
```

例 2: 外部ファイル読み込み時の例外の確認

次の例では、Java IO クラスを使用して DATA ステップから外部ファイルを読み込みます。この Java コードは **DataInputStream** のラッパークラスを作成し、**FileInputStream** をコンストラクターに渡せるようにします。コンストラクターが実際に取得するのは、**FileInputStream** の親である **InputStream** です。現在のメソッドルックアップはスーパークラスルックアップを実行できるほど堅牢でないため、ラッパーが必要です。

```
/* Java code */
public class myDataInputStream extends java.io.DataInputStream
{
  myDataInputStream(java.io.FileInputStream fi)
  {
    super(fi);
  }
}
```

作成したラッパークラスを使って外部ファイルの **DataInputStream** を作成し、ファイルの終端に達するまでファイルを読み込みます。EXCEPTIONCHECK メソッドを使って **readInt** メソッドでの **EOFException** 発生を判定し、入力ループを終了させることができます。

```
/* DATA step code */
data _null_;
  length d e 8;
```

```
  dcl javaobj f("java/io/File", "c:\temp\binint.txt");
  dcl javaobj fi("java/io/FileInputStream", f);
  dcl javaobj di("myDataInputStream", fi);
  do while(1);
    di.callIntMethod("readInt", d);
    di.ExceptionCheck(e);
    if (e) then
      leave;
    else
      put d=;
  end;
run;
```

関連項目

メソッド:

- “EXCEPTIONCHECK メソッド” (135 ページ)
- “EXCEPTIONDESCRIBE メソッド” (139 ページ)

EXCEPTIONDESCRIBE メソッド

デバッグログのオン/オフを切り替え、例外情報を出力します。

カテゴリ: 例外

適用対象: Java オブジェクト

構文

object.**EXCEPTIONDESCRIBE** (*status*);

引数

object
Java オブジェクト名を指定します。

status
デバッグログのオン/オフを指定します。**status** 引数は次のいずれかの値です。

0
デバッグログをオフに指定します。

1
デバッグログをオンに指定します。

デフォルト 0(オフ)

ヒント　Java から返されるステータス値のデータ型は DOUBLE です。これは SAS の数値データ値に対応します。

詳細

EXCEPTIONDESCRIBE メソッドは、例外のデバッグログのオンとオフを切り替えます。例外のデバッグログがオンの場合、例外情報が JVM の標準出力に出力されます。

注: デフォルトでは、JVM の標準出力は SAS ログにリダイレクトされます。

例: 例外情報の標準出力への出力

次の例では、例外情報は標準出力に出力されます。

```
/* Java code */
public class a
  {
    public void m() throws NullPointerException
    {
      throw new NullPointerException();
    }
  }

/* DATA step code */
data _null_;
  length e 8;
  dcl javaobj j('a');
  j.exceptiondescribe(1);
  rc = j.callvoidmethod('m');
run;
```

次の行が SAS ログに書き出されます。

```
java.lang.NullPointerException
    at a.m(a.java:5)
```

関連項目

メソッド:

- “EXCEPTIONCHECK メソッド” (135 ページ)
- “EXCEPTIONCLEAR メソッド” (137 ページ)

FLUSHJAVAOUTPUT メソッド

出力先に Java 出力を送信するように指定します。

カテゴリ: 出力

適用対象: Java オブジェクト

構文

object.**FLUSHJAVAOUTPUT**( );

引数

object
Java オブジェクト名を指定します。

詳細

SAS ログに対する Java 出力は、DATA ステップの終了時にフラッシュされます。FLUSHJAVAOUTPUT メソッドを使用した場合、Java 出力は DATA ステップの実行中に発行されたすべての出力の後に表示されます。

例: Java 出力の表示

次の例では、DATA ステップの完了後に"In Java class"行が書き込まれます。

```
/* Java code */
 public class p
{
  void p()
    {
      System.out.println("In Java class");
    }
}

/* DATA step code */
data _null_;
  dcl javaobj j('p');
  do i = 1 to 3;
     j.callVoidMethod('p');
     put 'In DATA Step';
  end;
run;
```

次の行が SAS ログに書き込まれます。

```
In DATA Step
In DATA Step
In DATA Step
In Java class
In Java class
In Java class
```

FLUSHJAVAOUTPUT メソッドを使用した場合、Java 出力が実行順に SAS ログに書き込まれます。

```
/* DATA step code */
data _null_;
  dcl javaobj j('p');
  do i = 1 to 3;
     j.callVoidMethod('p');
     j.flushJavaOutput();
     put 'In DATA Step';
  end;
run;
```

次の行が SAS ログに書き出されます。

```
In Java class
In DATA Step
In Java class
In DATA Step
In Java class
In DATA Step
```

関連項目

"Java 標準出力"

GETtypeFIELD メソッド

Java オブジェクトの静的でないフィールドの値を返します。

カテゴリ: フィールド参照

適用対象: Java オブジェクト

構文

object.**GET***type*__FIELD__ ("*field-name*", *value*);

引数

object
Java オブジェクト名を指定します。

type
Java フィールドの型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
フィールドの型に BOOLEAN を指定します。

BYTE
フィールドの型に BYTE を指定します。

CHAR
フィールドの型に CHAR を指定します。

DOUBLE
フィールドの型に DOUBLE を指定します。

FLOAT
フィールドの型に FLOAT を指定します。

INT
フィールドの型に INT を指定します。

LONG
フィールドの型に LONG を指定します。

SHORT
フィールドの型に SHORT を指定します。

STRING
フィールドの型に STRING を指定します。

参照項目: “型の問題”

field-name
Java フィールド名を指定します。

要件 フィールド名は単一引用符または二重引用符で囲む必要があります。

value
返されたフィールド値を受け取るする変数名を指定します。

詳細

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、パブリックフィールドにアクセスして編集できます。GET*type*FIELD メソッドでは、静的でないフィールドにアクセスできます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、“型の問題”を参照してください。

比較

GET*type*FIELD メソッドは、Java オブジェクトの静的でないフィールドの値を返します。静的なフィールドの値を返すには、GETSTATIC*type*FIELD メソッドを使用します。

例: 静的でないフィールドの値の取得

次の例では、3 つの静的でないフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、GET*type*FIELD メソッドを使用してフィールドの値が変更および取得されます。

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttest
{
  public int i;
  public double d;
  public string s;
}
}
```

```
/*  DATA step code */
data _null_;
  dcl javaobj j("ttest");
  length val 8;
  length str $20;
  j.setIntField("i", 100);
  j.setDoubleField("d", 3.14159);
  j.setStringField("s", "abc");
  j.getIntField("i", val);
  put val=;
  j.getDoubleField("d", val);
  put val=;
  j.getStringField("s", str);
  put str=;
run;
```

次の行が SAS ログに書き出されます。

```
val=100
val=3.14159
str=abc
```

関連項目

メソッド:

- “GETSTATICtypeFIELD メソッド” (145 ページ)
- “SETtypeFIELD メソッド” (149 ページ)

GETSTATICtypeFIELD メソッド

Java オブジェクトの静的なフィールドの値を返します。

カテゴリ: フィールド参照

適用対象: Java オブジェクト

構文

object.**GETSTATIC***type***FIELD** ("*field-name*", *value*);

引数

object
Java オブジェクト名を指定します。

type
Java フィールドの型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
フィールドの型に BOOLEAN を指定します。

BYTE
フィールドの型に BYTE を指定します。

CHAR
フィールドの型に CHAR を指定します。

DOUBLE
フィールドの型に DOUBLE を指定します。

FLOAT
フィールドの型に FLOAT を指定します。

INT
フィールドの型に INT を指定します。

LONG
フィールドの型に LONG を指定します。

SHORT
フィールドの型に SHORT を指定します。

STRING
フィールドの型に STRING を指定します。

参照項目: “型の問題”

field-name
Java フィールド名を指定します。

要件 フィールド名は単一引用符または二重引用符で囲む必要があります。

value
返されたフィールド値を受け取るする変数名を指定します。

詳細

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、パブリックフィールドにアクセスして編集できます。GETSTATIC*type*FIELD メソッドでは、静的なフィールドにアクセスできます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、"型の問題"を参照してください。

比較

GETSTATIC*type*FIELD メソッドは、Java オブジェクトの静的なフィールドの値を返します。静的でないフィールドの値を返すには、GET*type*FIELD メソッドを使用します。

例: 静的なフィールドの値の取得

次の例では、3 つの静的なフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、GETSTATIC*type*FIELD メソッドを使用してフィールドの値が設定および取得されます。

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttest
{
  public int i;
  public double d;
  public string s;
}
}

/*  DATA step code */
data _null_;
  dcl javaobj j("ttest");
  length val 8;
  length str $20;
  j.setStaticIntField("i", 100);
  j.setStaticDoubleField("d", 3.14159);
  j.setStaticStringField("s", "abc");
  j.getStaticIntField("i", val);
  put val=;
```

```
  j.getStaticDoubleField("d", val);
  put val=;
  j.getStaticStringField("s", str);
  put str=;
run;
```

次の行が SAS ログに書き出されます。

```
val=100
val=3.14159
str=abc
```

関連項目

メソッド:

- “GETtypeFIELD メソッド” (142 ページ)
- “SETSTATICtypeFIELD メソッド” (151 ページ)

NEW 演算子: Java オブジェクト

Java オブジェクトのインスタンスを作成します。

該当要素: DATA ステップ

適用対象: Java オブジェクト

構文

object-reference = **_NEW_ JAVAOBJ** ("*java-class*", <*argument-1*, ...*argument-n*>);

引数

object-reference
Java オブジェクトのオブジェクト参照名を指定します。

java-class
インスタンス化する Java クラスの名前を指定します。

要件 Java クラス名は単一引用符または二重引用符で囲む必要があります。

argument
Java オブジェクトのインスタンスの作成に使用する情報を指定します。
argument の有効な値は Java オブジェクトによって異なります。

詳細

SAS プログラムで DATA ステップコンポーネントオブジェクトを使用するには、オブジェクトを宣言して作成(インスタンス化)する必要があります。DATA ステップコンポーネントインターフェイスは、DATA ステップ内から定義済みのコンポーネントオブジェクトにアクセスするメカニズムを提供します。

_NEW_演算子を使用して Java オブジェクトをインスタンス化する場合は、最初に DECLARE ステートメントを使用して Java オブジェクトを宣言する必要があります。たとえば、次のコード行では、DECLARE ステートメントは、オブジェクト参照 J が Java オブジェクトであることを SAS に示します。_NEW_演算子で Java オブジェクトが作成され、オブジェクト参照 J に割り当てられます。

```
declare javaobj j;
j = _new_ javaobj("somejavaclass" );
```

注: DECLARE ステートメントを使用して Java オブジェクトの宣言とインスタンス化を 1 つのステップで実行できます。

コンストラクターは、コンポーネントオブジェクトをインスタンス化し、コンポーネントオブジェクトデータを初期化するために使用するメソッドです。たとえば、次のコード行では、_NEW_演算子で Java オブジェクトがインスタンス化され、オブジェクト参照 J に割り当てられます。Java オブジェクトコンストラクターで唯一必要な引数は、インスタンス化する Java クラスの名前です。その他の引数はすべて Java クラス自体のコンストラクター引数です。次の例では、Java クラス名 **testjavaclass** がコンストラクターで、値 **100** と **.8** はコンストラクター引数です。

```
declare javaobj j;
j = _new_ javaobj("testjavaclass", 100, .8);
```

定義済みの DATA ステップコンポーネントオブジェクトおよびコンストラクターの詳細については、“コンポーネントオブジェクト”を参照してください。

比較

Java オブジェクトを宣言し、インスタンスを作成するには、DECLARE ステートメントと_NEW_演算子を使用するか、DECLARE ステートメントのみを使用します。

例: _NEW_演算子を使用した Java クラスのインスタンス化と初期化

次の例では、ハッシュテーブルの Java クラスが作成されます。_NEW_演算子を使用して、容量および負荷係数を指定し、このクラスのインスタンスを作成します。この例では、DATA ステップの唯一の数値型が Java のデータ型 DOUBLE と同等であるため、ラッパークラス **mhash** が必要です。

```
/* Java code */
import java.util.*;
public class mhash extends Hashtable;
```

```
{
  mhash (double size, double load)
    {
      super ((int)size, (float)load);
    }
}

/* DATA step code */
data _null_;
  declare javaobj h;
  h = _new_ javaobj("mhash", 100, .8);
run;
```

関連項目

ステートメント:

- “DECLARE ステートメント: Java オブジェクト” (132 ページ)

SETtypeFIELD メソッド

Java オブジェクトの静的でないフィールドの値を変更します。

カテゴリ: フィールド参照

適用対象: Java オブジェクト

構文

object.**SET***type***FIELD** ("*field-name*", *value*);

引数

object
Java オブジェクト名を指定します。

type
Java フィールドの型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
フィールドの型に BOOLEAN を指定します。

BYTE
フィールドの型に BYTE を指定します。

CHAR
フィールドの型に CHAR を指定します。

DOUBLE
フィールドの型に DOUBLE を指定します。

FLOAT
フィールドの型に FLOAT を指定します。

INT
フィールドの型に INT を指定します。

LONG
フィールドの型に LONG を指定します。

SHORT
フィールドの型に SHORT を指定します。

STRING
フィールドの型に STRING を指定します。

参照項目: "型の問題"

field-name
Java フィールド名を指定します。

要件 フィールド名は単一引用符または二重引用符で囲む必要があります。

value
フィールドの値を指定します。

詳細

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、パブリックフィールドにアクセスして編集できます。SET*type*FIELD メソッドを使用すると、静的でないフィールドの値を変更できます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、"型の問題"を参照してください。

比較

SET*type*FIELD メソッドは、Java オブジェクトの静的でないフィールドの値を変更します。静的フィールドの値を変更するには、SETSTATIC*type*FIELD メソッドを使用します。

例: 静的でないフィールドを持つ Java クラスの作成

次の例では、3 つの静的でないフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、SET*type*FIELD メソッドでフィールド値が設定されてから、フィールド値が取得されます。

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttest
{
  public int i;
  public double d;
  public string s;
}
}
```

```
/*  DATA step code */
data _null_;
  dcl javaobj j("ttest");
  length val 8;
  length str $20;
  j.setIntField("i", 100);
  j.setDoubleField("d", 3.14159);
  j.setStringField("s", "abc");
  j.getIntField("i", val);
  put val=;
  j.getDoubleField("d", val);
  put val=;
  j.getStringField("s", str);
  put str=;
run;
```

次の行が SAS ログに書き出されます。

```
val=100
val=3.14159
str=abc
```

関連項目

メソッド:

- “GETtypeFIELD メソッド” (142 ページ)
- “SETSTATICtypeFIELD メソッド” (151 ページ)

SETSTATICtypeFIELD メソッド

Java オブジェクトの静的フィールドの値を変更します。

カテゴリ: フィールド参照

適用対象: Java オブジェクト

構文

object.**SETSTATIC***type***FIELD** ("*field-name*", *value*);

引数

object
Java オブジェクト名を指定します。

type
Java フィールドの型を指定します。型には次のいずれかの値を指定できます。

BOOLEAN
フィールドの型に BOOLEAN を指定します。

BYTE
フィールドの型に BYTE を指定します。

CHAR
フィールドの型に CHAR を指定します。

DOUBLE
フィールドの型に DOUBLE を指定します。

FLOAT
フィールドの型に FLOAT を指定します。

INT
フィールドの型に INT を指定します。

LONG
フィールドの型に LONG を指定します。

SHORT
フィールドの型に SHORT を指定します。

STRING
フィールドの型に STRING を指定します。

参照項目: “型の問題”

field-name
Java フィールド名を指定します。

要件 フィールド名は単一引用符または二重引用符で囲む必要があります。

value
フィールドの値を指定します。

詳細

Java オブジェクトをインスタンス化したら、その Java オブジェクトのメソッド呼び出しを使用し、パブリックフィールドにアクセスして編集できます。
SETSTATIC*type*FIELD メソッドを使用すると、静的フィールドの値を変更できます。

注: *type* 引数は Java のデータ型を表します。Java のデータ型と SAS のデータ型の関連性の詳細については、"型の問題"を参照してください。

比較

SETSTATIC*type*FIELD メソッドは、Java オブジェクトの静的フィールドの値を変更します。静的でないフィールドの値を変更するには、SET*type*FIELD メソッドを使用します。

例: 静的フィールドを持つ Java クラスの作成

次の例では、3 つの静的なフィールドを含む単純なクラスを作成します。Java オブジェクト **j** がインスタンス化され、SETSTATIC*type*FIELD メソッドでフィールド値が設定されてから、フィールド値が取得されます。

```
/* Java code */
import java.util.*;
import java.lang.*;
public class ttestc
{
  public static double d;
  public static double dm()
  {
    return d;
  }
}

/*  DATA step code */
data _null_;
  dcl javaobj j("ttest");
  length val 8;
  length str $20;
  j.setStaticIntField("i", 100);
  j.setStaticDoubleField("d", 3.14159);
  j.setStaticStringField("s", "abc");
  j.getStaticIntField("i", val);
  put val=;
  j.getStaticDoubleField("d", val);
  put val=;
  j.getStaticStringField("s", str);
  put str=;
run;
```

次の行が SAS ログに書き出されます。

```
val=100
val=3.14159
str=abc
```

関連項目

メソッド:

- “GETSTATICtypeFIELD メソッド” (145 ページ)
- “SETtypeFIELD メソッド” (149 ページ)

3 部

FCMP コンポーネントオブジェクト

6 章
PROC FCMP ハッシュオブジェクトとハッシュ反復子オブジェクトの使用 . 157

7 章
PROC FCMP ハッシュおよびハッシュ反復子言語要素 159

8 章
ディクショナリオブジェクトの使用 219

9 章
ディクショナリオブジェクトの言語要素 227

10 章
Python オブジェクトの使用 251

11 章
Python オブジェクトの言語要素 263

6

PROC FCMP ハッシュオブジェクトとハッシュ反復子オブジェクトの使用

PROC FCMP ハッシュオブジェクトと PROC FCMP ハッシュ反復子オブジェクトの使用 157

PROC FCMP ハッシュオブジェクトと PROC FCMP ハッシュ反復子オブジェクトの使用

PROC FCMP ハッシュオブジェクトと PROC FCMP ハッシュ反復子オブジェクトは、ルックアップキーに基づくデータの格納、検索、フィルタリング、および取得を迅速かつ効率的に行うことができます。これらのオブジェクトは、属性、メソッド、および演算子で構成されるデータ要素です。属性とは、オブジェクトに関連付けられた情報を指定するプロパティです。メソッドは、オブジェクトが実行できる演算を定義します。演算子は特別な機能を提供します。コンポーネントオブジェクトの属性とメソッドにアクセスするには、DATA ステップオブジェクトのドット表記を使用します。

ハッシュ関数を用いて、入力された文字列や数値(キー)を整数(ハッシュ)に変換します。このハッシュ値は、ハッシュテーブルのインデックスとして使用されます。各キーに対して、ハッシュテーブルは関連するデータを格納し、取得できます。ハッシュ化は、キーで参照される大量の情報を検索するための最速の方法とされています。

PROC FCMP のハッシュオブジェクトとハッシュ反復子オブジェクトでサポートされているステートメント、メソッド、および構文の一覧については、PROC FCMP ハッシュとハッシュ反復子の言語要素を参照してください。

PROC FCMP でのハッシュの使い方や例題の確認については、PROC FCMP のハッシュ化で生産性を高めるを参照してください。

PROC FCMP とハッシュオブジェクトを使った別の例については、PROC FCMP を使って SAS データセットをハッシュオブジェクトに読み込むを参照してください。

7

PROC FCMP ハッシュおよびハッシュ反復子言語要素

ディクショナリ **160**
ADD メソッド 160
CHECK メソッド 162
CLEAR メソッド 164
DEFINEDATA メソッド 165
DEFINEDONE メソッド 167
DEFINEKEY メソッド 169
DELETE メソッド: ハッシュオブジェクトとハッシュ反復子オブジェクト 171
DO_OVER メソッド 171
EQUALS メソッド 173
FIND メソッド 176
FIND_NEXT メソッド 178
FIND_PREV メソッド 180
HAS_NEXT メソッド 181
HAS_PREV メソッド 183
DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト 185
NUM_ITEMS メソッド 属性 190
REF メソッド 192
REMOVEDUP メソッド 194
REMOVE メソッド 197
REPLACE メソッド 199
REPLACEDUP メソッド 201
RESET_DUP メソッド 204
SUMDUP メソッド 205
SUM メソッド 208
FIRST メソッド 210
LAST メソッド 212
NEXT メソッド 213
PREV メソッド 215
SETCUR メソッド 216

ディクショナリ

ADD メソッド

PROC FCMP ハッシュオブジェクトにキーと値のペアを追加します。

適用対象: PROC FCMP ハッシュオブジェクト

構文

rc=*object*.**ADD** (<<KEY: *keyvalue-1*>, ...<KEY: *keyvalue-n*>,
<DATA: *datavalue-1* >, ...<DATA: *datavalue-n*>>);

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。*keyvalue* はリテラルでなければなりません。

“KEY: *keyvalue*”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。*datavalue* はリテラルでなければなりません。

“DATA: *datavalue*”ペアの数は、DEFINEDATA メソッドを使用して定義されたデータ変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで ADD メソッドを使用して、ハッシュオブジェクト内にデータを保存できます。

1 つ目は、次のコードに示すように、キーおよびデータアイテムを 定義してから ADD メソッドを使用する方法です。

```
proc fcmp;
  length k $8;
  length d $12;
  /* Declare hash object and key and data variable names */
     declare hash h();
     rc = h.defineKey('k');
     rc = h.defineData('d');
     rc = h.defineDone();
  /* Define constant key and data values */
  k = 'Joyce';
  d = 'Ulysses';
  /* Add key and data values to hash object */
  rc = h.add();
run;
```

すでにハッシュオブジェクト内にあるキーを追加した場合、ADD メソッドはゼロ以外の値を返します。指定したキーに関連付けられたデータアイテムを新しいデータに置き換えるには、REPLACE メソッドを使用します。

DEFINEDATA メソッドでデータ変数を指定しない場合、データ変数は自動的にキーと同じであるとみなされます。

2 つ目は、次のコードに示すように、ショートカットを使用して、ADD メソッド呼び出しでキーおよびデータアイテムを 直接指定する方法です。

```
proc fcmp;
  length k $8;
  length d $12;
  /* Define hash object and key and data variable names */
     declare hash h();
     rc = h.defineKey('k');
     rc = h.defineData('d');
     rc = h.defineDone();
     /* avoid uninitialized variable notes */
     call missing(k, d);
  /* Define constant key and data values and add to hash object */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
run;
```

KEY および DATA 引数タグを使用して直接キーとデータを指定する場合、両方の引数タグを使用する必要があります。

ADD メソッドでは、データ変数の値はデータアイテムの値に設定されません。ADD メソッドはハッシュオブジェクトの値のみを設定します。キー変数またはデータ変数への割り当てがこのプログラムに出現しないため、変数が初期化されていないことを示す NOTE が発行されます。これらの NOTE を受け取らないようにするには、CALL MISSING ルーチンを使用して、キーとデータ変数をパラメーターとして指定します。

関連項目

- “ハッシュオブジェクトの使用”

メソッド:

- “DEFINEDATA メソッド” (165 ページ)
- “DEFINEKEY メソッド” (169 ページ)

CHECK メソッド

指定したキーが PROC FCMP ハッシュオブジェクト内に保存されているかどうかを確認します。

適用対象: PROC FCMP ハッシュオブジェクト

構文

rc=*object*.**CHECK** (<KEY: *keyvalue-1*, ...KEY: *keyvalue-n*>);

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。*keyvalue* はリテラルでなければなりません。

“KEY: *keyvalue*”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで CHECK メソッドを使用して、ハッシュオブジェクト内のデータを検索できます。

1 つ目は、次のコードに示すように、キーを指定してから CHECK メソッドを使用する方法です。

```
proc fcmp;
  length k $8;
  length d $12;
  /* Declare hash object and key and data variable names */
    declare hash h();
    rc = h.defineKey('k');
```

```
    rc = h.defineData('d');
    rc = h.defineDone();

 /* avoid uninitialized variable notes */
    call missing(k, d);
  /* Define constant key and data values and add to hash object */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  /* Verify that JOYCE key is in hash object */
  k = 'Joyce';
  rc = h.check();
  if (rc = 0) then
    put 'Key is in the hash object.';
run;
```

2 つ目は、次のコードに示すように、ショートカットを使用して、CHECK メソッド呼び出しでキーを直接指定する方法です。

```
proc fcmp;
  length k $8;
  length d $12;
  /* Declare hash object and key and data variable names */
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();

 /* avoid uninitialized variable notes */
    call missing(k, d);
  /* Define constant key and data values and add to hash object */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  /* Verify that JOYCE key is in hash object */
  rc = h.check(key: 'Joyce');
  if (rc  =0) then
    put 'Key is in the hash object.';
run;
```

比較

CHECK メソッドは、キーがハッシュオブジェクト内にあるかどうかを示す値のみを返します。キーに関連付けられたデータ変数は更新されません。FIND メソッドも、キーがハッシュオブジェクト内にあるかどうかを示す値を返します。ただし、キーがハッシュオブジェクト内にある場合、FIND メソッドはさらにデータ変数をデータアイテムの値に設定し、メソッド呼び出し後にそのデータアイテムを使用できるようにします。

関連項目

メソッド:

- “DEFINEKEY メソッド” (169 ページ)
- “FIND メソッド” (176 ページ)

CLEAR メソッド

PROC FCMP ハッシュオブジェクトインスタンスを削除せずに、ハッシュオブジェクトからすべてのアイテムを削除します。

適用対象: PROC FCMP ハッシュオブジェクト

構文

rc=*object*.**CLEAR** ();

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

詳細

CLEAR メソッドでは、既存のハッシュオブジェクトからアイテムを削除して、オブジェクトを削除して新規作成することなく再利用できます。ハッシュオブジェクトインスタンスを完全に削除する場合は、DELETE メソッドを使用します。

注: CLEAR メソッドは、変数の値を変更しません。ハッシュオブジェクトの値のみをクリアします。

例: ハッシュオブジェクトのクリア

この例では、ハッシュオブジェクトを宣言し、ハッシュオブジェクト内のアイテム数を取得してから、ハッシュオブジェクトを削除せずにクリアします。

```
proc fcmp ;

/* Declares the hash object named h */
  declare hash h1();
  rc= h1.defineKey('key1');
  rc=h1.defineData('key1');
```

```
  rc=h1.defineDone();


  key1 = 'abc';
  rc= h1.add();
/* Uses the NUM_ITEMS attribute, which returns the */
/* number of items in the hash object.          */
  n = h1.num_items();
  put 'Number of items before clear = ' n;
/* Uses the CLEAR method to delete all items within h. */
  rc = h1.clear();
/* Writes the number of items in the log. */
  n = h1.num_items();
  put 'Number of items after clear = ' n;
 quit;
 run;
```

上記のステートメントは、次の結果を生成します。

```
Number of items before clear = 1
Number of items after clear = 0
```

関連項目

メソッド:

- “DELETE メソッド: ハッシュオブジェクトとハッシュ反復子オブジェクト” (171 ページ)

DEFINEDATA メソッド

指定のデータ変数に関連付けられたデータを、PROC FCMP ハッシュオブジェクトに格納するよう定義します。

適用対象: PROC FCMP ハッシュオブジェクト

構文

rc=*object*.**DEFINEDATA** ('*datavarname-1*' <'*datavarname-2*', ...>);

rc=*object*.**DEFINEDATA** (ALL: 'YES'" | YES");

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

'*datavarname*'
データ変数の名前を指定します。

データ変数名は二重引用符で囲むこともできます。

ALL:'YES' | “YES”
データセットをオブジェクトコンストラクターに読み込む場合に、すべてのデータ変数をデータとして指定します。

DECLARE ステートメントで *dataset* 引数タグを使用してデータセットを自動的に読み込む場合、ALL: 'YES'オプションを使ってすべてのデータセット 変数をデータとして定義できます。

詳細

ハッシュオブジェクトは、ルックアップキーに基づくデータの保存と取得によって機能します。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。ハッシュオブジェクトの初期化を完了するには、すべてのキー変数とデータ変数を定義してから、DEFINEDONE メソッドを呼び出す必要があります。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

注: PROC FCMP は DEFINEDATA メソッドを複数回実行しません。DATA ステップの DEFINEDATA メソッドとは異なり、PROC FCMP で DEFINEDATA メソッドが繰り返し実行されないように、条件付きの do ブロックを作成する必要はありません。

注: ADD メソッドまたは REPLACE メソッドにショートカット表記(例:**h.add(key:99, data:'apple', data:'orange')**)を使い、DEFINEDATA メソッドに ALL: 'YES'オプションを使う場合は、データの指定順序をデータセット内と同じにする必要があります。

注: ハッシュオブジェクトは値をキー変数(例:**h.find(key:'abc')** >)に割り当てないため、ハッシュオブジェクトとハッシュ反復子が実行するキーとデータの変数割り当てを、SAS コンパイラからは検出できません。したがって、キー変数またはデータ変数への割り当てがプログラムに出現しない場合、変数が初期化されていないことを示す NOTE が発行されます。このような NOTE が発行されないようにするには、次のアクションのいずれかを実行します。

- NONOTES システムオプションを設定します。
- 各キー変数およびデータ変数について、初期割り当てステートメントを(通常は欠損値に)指定します。

- CALL MISSING ルーチンを、すべてのキー変数とデータ変数をパラメーターとして使用します。

DEFINEDATA メソッドの使用方法の詳細については、"キーとデータの定義"を参照してください。

例

この例では、ハッシュオブジェクトを作成し、キー変数とデータ変数を定義します。

```
proc fcmp;
  length d $20;
  length k $20;
  /* Declare the hash object and key and data variables */
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
run;
```

関連項目

- "キーとデータの定義"

メソッド:

- "DEFINEDONE メソッド" (167 ページ)
- "DEFINEKEY メソッド" (169 ページ)

ステートメント:

- "DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト" (185 ページ)

DEFINEDONE メソッド

PROC FCMP ハッシュオブジェクトのすべてのキーとデータの定義が完了したことを示します。

適用対象: PROC FCMP ハッシュオブジェクト

構文

rc=*object*.**DEFINEDONE**( );

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

詳細

DEFINEDONE メソッド呼び出しでコンストラクターに *dataset* 引数タグを使用すると、データセットはハッシュオブジェクトに読み込まれます。

ハッシュオブジェクトは、ルックアップキーに基づくデータの保存と取得によって機能します。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。ハッシュオブジェクトの初期化を完了するには、すべてのキー変数とデータ変数を定義してから、DEFINEDONE メソッドを呼び出す必要があります。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

注: DATA ステップの DEFINEDATA メソッドとは異なり、PROC FCMP で DEFINEDATA メソッドが繰り返し実行されないように、条件付きの do ブロックを作成する必要はありません。

DEFINEDONE メソッドの使用方法の詳細については、“キーとデータの定義”を参照してください。

関連項目

- “キーとデータの定義”

メソッド:

- “DEFINEDATA メソッド” (165 ページ)
- “DEFINEKEY メソッド” (169 ページ)

DEFINEKEY メソッド

PROC FCMP ハッシュオブジェクトのキー変数を定義します。

適用対象: PROC FCMP ハッシュオブジェクト

構文

rc=*object*.**DEFINEKEY**('*keyvarname-1* '<, ...'*keyvarname-2*', ...>);

rc=*object*.**DEFINEKEY**(ALL: 'YES'" | YES");

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

'keyvarname'
キー変数の名前を指定します。

キー変数名は二重引用符で囲むこともできます。

ALL:'YES' | "YES"
データセットをオブジェクトコンストラクターに読み込む場合に、すべてのデータ変数をキーとして指定します。

DECLARE ステートメントまたは_NEW_演算子で *dataset* 引数タグを使用してデータセットを自動的に読み込む場合、ALL: 'YES'オプションを使ってすべてのキー変数を定義できます。

詳細

ハッシュオブジェクトは、ルックアップキーに基づくデータの保存と取得によって機能します。キーとデータは、ドット表記のメソッド呼び出しでハッシュオブジェクトを初期化するときに使う DATA ステップ変数です。キーを定義するには、キー変数名を DEFINEKEY メソッドに渡します。データを定義するには、データ変数名を DEFINEDATA メソッドに渡します。ハッシュオブジェクトの初期化を完了するには、すべてのキー変数とデータ変数を定義してから、DEFINEDONE メソッドを呼び出す必要があります。キーとデータには、任意の数の文字または数値 DATA ステップ変数を使用できます。

DEFINEKEY メソッドの使用方法の詳細については、“キーとデータの定義”を参照してください。

注: PROC FCMP は DEFINEKEY メソッドを複数回実行しません。DATA ステップの DEFINEKEY メソッドとは異なり、PROC FCMP で DEFINEKEY メソッドが繰り返し実行されないように、条件付きの do ブロックを作成する必要はありません。

注: ADD、CHECK、FIND、REMOVE、REPLACE などのメソッドにショートカット表記(例:**h.add(key:99, data:'apple', data:'orange')**)を使い、DEFINEKEY メソッドに ALL: 'YES'オプションを使う場合は、キーとデータの指定順序をデータセット内と同じにする必要があります。

注: ハッシュオブジェクトは値をキー変数(例:**h.find(key:'abc')** >)に割り当てないため、ハッシュオブジェクトとハッシュ反復子が実行するキーとデータの変数割り当てを、SAS コンパイラからは検出できません。したがって、キー変数またはデータ変数への割り当てがプログラムに出現しない場合、変数が初期化されていないことを示す NOTE が発行されます。このような NOTE が発行されないようにするには、次のアクションのいずれかを実行します。

- NONOTES システムオプションを設定します。
- 各キー変数およびデータ変数について、初期割り当てステートメントを(通常は欠損値に)指定します。
- CALL MISSING ルーチンを、すべてのキー変数とデータ変数をパラメーターとして使用します。次に例を示します。

```
length d $20;
length k $20;
  declare hash h();
  rc = h.defineKey('k');
  rc = h.defineData('d');
  rc = h.defineDone();
  call missing(k, d);
```

関連項目

- “キーとデータの定義”

メソッド:

- “DEFINEDATA メソッド” (165 ページ)
- “DEFINEDONE メソッド” (167 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (185 ページ)

DELETE メソッド: ハッシュオブジェクトとハッシュ反復子オブジェクト

PROC FCMP ハッシュオブジェクトまたは PROC FCMP ハッシュ反復子オブジェクトを削除します。

適用対象: PROC FCMP ハッシュオブジェクト, PROC FCMP ハッシュ反復子オブジェクト

構文

rc=*object*.**DELETE**( );

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに出力されます。

object
ハッシュオブジェクトまたはハッシュ反復子オブジェクトの名前を指定します。

詳細

PROC FCMP コンポーネントオブジェクトは、PROC ステップの終了時に自動的に削除されます。他のハッシュオブジェクトまたはハッシュ反復子オブジェクトでオブジェクト参照名を再利用する場合、またはそのオブジェクトで使用されるメモリをリリースする場合は、DELETE メソッドでハッシュオブジェクトまたはハッシュ反復子オブジェクトを削除します。

削除したハッシュオブジェクトまたはハッシュ反復子オブジェクトを使用しようとすると、エラーがログに書き込まれます。

ハッシュオブジェクト内からすべてのアイテムを 削除し、ハッシュオブジェクトを再利用できるように保存するには、CLEAR メソッドを使用します。

DO_OVER メソッド

ハッシュオブジェクトの重複キーのリスト内を移動します。

適用対象: PROC FCMP ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

object.**DO_OVER** (<KEY:*keyvalue*>);

引数

object
ハッシュオブジェクト名を指定します。

KEY:*keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

詳細

ハッシュオブジェクトが単一のキーに対して複数の値を持つ場合、DO_OVER メソッドを DO ループの反復内で使用して、重複キー間を移動します。DO_OVER メソッドは最初のメソッド呼び出しでキーを読み込み、キーが最後に到達するまで重複キーリストを移動しつづけます。

注: 反復の途中でキーを切り替えた場合、RESET_DUP メソッドを使用してポインターをリストの最初にリセットしなければなりません。そうしないと、SAS は最初のキーを使い続けます。

例

次の例では、重複キーを含むデータセット **dup** を作成します。DO_OVER メソッドと RESET_DUP メソッドを使って、重複キー間を移動します。

```
data dup;
  input key_id value;
  datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
run;
```

```
proc fcmp;
file log;
  dcl hash h(dataset:'dup', multidata: 'y', ordered: 'y');
  rc= h.definekey('key_id');
  rc= h.definedata('key_id', 'value');
  rc= h.definedone();
  call missing(key_id, value);

  rc= h.reset_dup();
  key_id = 2;
  do while(h.do_over(key:2) eq 0);
    put key_id= value=;
  end;

  key_id = 3;
  do while(h.do_over(key:3) eq 0);
    put key_id= value=;
  end;

  key_id = 2;
  do while(h.do_over(key:2) eq 0);
    put key_id= value=;
  end;
run;
quit;
```

次の行が SAS ログに書き出されます。

```
key_id=2 value=11
key_id=2 value=16
key_id=2 value=9
key_id=3 value=20
key_id=3 value=100
key_id=2 value=11
key_id=2 value=16
key_id=2 value=9
```

関連項目

メソッド:

- “RESET_DUP メソッド”

EQUALS メソッド

2 つのハッシュオブジェクトが等しいかどうかを確認します。

適用対象: PROC FCMP ハッシュオブジェクト

注: PROC FCMP EQUALS メソッドは 2024.07 から使用可能です。

構文

rc=*object*.**EQUALS** (HASH: '*object*', RESULT: *variable _name*);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

HASH:'*object*'
1 つ目のハッシュオブジェクトに対して比較する 2 つ目のハッシュオブジェクトの名前を指定します。

RESULT: *variable_name*
結果を保持する数値変数を指定します。ハッシュオブジェクトが等しい場合、結果変数は 1 です。それ以外の場合の結果変数はゼロです。

詳細

次の例では、H1 と H2 のハッシュオブジェクトを比較しています。

```
proc fcmp;
file log
   declare hash h1();
   rc=h1.defineKey('k');
   rc=h1.defineDone();

   declare hash h2();
   rc=h2.defineKey('k');
   rc=h2.defineDone();
   call missing(k);

   rc = h1.equals(hash: 'h2', result: eq);
   if eq then
     put 'hash objects are equal';
   else
     put 'hash objects are not equal';
run;
```

次の行が SAS ログに書き込まれます。

```
ハッシュオブジェクトが等しい
```

これらの条件がすべて揃ったとき、2 つのハッシュオブジェクトは等しいと定義されます。

- 2 つのハッシュオブジェクトが同サイズである(HASHEXP サイズが等しい)。

- 2 つのハッシュオブジェクトに同数のアイテムがある(H1.NUM_ITEMS = H2.NUM_ITEMS)。
- 2 つのハッシュオブジェクトのキー構造とデータ構造が同一である。
- H1 と H2 ハッシュオブジェクトの比較を順不同で反復した場合に、H1 からの一連のレコードが対応する H2 のレコードと同じキーおよびデータのフィールドを持っている。すなわち、各ハッシュオブジェクトの同じ位置に各レコードがあり、それらのレコードが対応するもう一方のハッシュオブジェクトのレコードと同一である。

例: 2 つのハッシュオブジェクトの比較

次の例では、H1 と H2 のハッシュオブジェクトが宣言され、インスタンス化されています。キーの値が 99 のレコードが両方のオブジェクトに追加されます。EQUALS の最初の呼び出しは、ハッシュオブジェクトが等価であることを示す 0 以外の結果値を返します。

そして、H2 のキーを 100 という値に置き換えます。2 回目の EQUALS の呼び出しでは、H1 と H2 のハッシュオブジェクトを比較し、ハッシュオブジェクトが等価でなくなったことを示す 0 の結果値を返します。

```
proc fcmp;
file log;
 declare hash h1();
  rc=h1.defineKey('k');
  rc=h1.defineDone();

  declare hash h2();
  rc=h2.defineKey('k');
  rc=h2.defineDone();

  k = 99;
  rc=h1.add();
  rc=h2.add();
  rc = h1.equals(hash: 'h2', result: eq);
  put eq=;

  k = 100;
  rc=h2.replace();

  rc = h1.equals(hash: 'h2', result: eq);
  put eq=;
 run;
```

次の行が SAS ログに書き出されます。

```
eq=1
eq=0
```

FIND メソッド

指定されたキーが PROC FCMP ハッシュオブジェクトに格納されているかどうかを判断し、見つかった場合はデータセットのデータ変数を更新します。

適用対象: PROC FCMP ハッシュオブジェクト

構文

rc=*object*.**FIND** (<KEY: *keyvalue-1*, KEY: *keyvalue-2*, ...>);

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。*keyvalue* はリテラルでなければなりません。

"KEY: *keyvalue*"ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで FIND メソッドを使用して、ハッシュオブジェクト内のデータを検索できます。

1 つ目は、このコードに示すように、キーを指定してから FIND メソッドを使用する方法です。

```
proc fcmp;
  length k $8  d $12;
  /* Declare hash object and key and data variables */
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
```

```
  end;
  /* Define constant key and data values */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  /* Find the key JOYCE */
  k = 'Joyce';
  rc = h.find();
  if (rc = 0) then
     put 'Key is in the hash object.';
run;
```

2 つ目は、このコードに示すように、ショートカットを使用して、FIND メソッド呼び出しでキーを直接指定する方法です。

```
proc fcmp;
  length k $8  d $12;
  /* Declare hash object and key and data variables */
  if _N_ = 1 then do;
     declare hash h();
     rc = h.defineKey('k');
     rc = h.defineData('d');
     rc = h.defineDone();
     /* avoid uninitialized variable notes */
     call missing(k, d);
  end;
   /* Define constant key and data values */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
 /* Find the key JOYCE */
  rc = h.find(key: 'Joyce');
  if (rc = 0) then
     put 'Key is in the hash object.';
run;
```

比較

FIND メソッドは、キーがハッシュオブジェクト内に格納されているかどうかを示す値を返します。FIND メソッドでは、キーがハッシュオブジェクトに存在する場合、同時にデータ変数がデータアイテムの値に設定され、メソッド呼び出し後に値が使えるようになります。CHECK メソッドは、キーがハッシュオブジェクトに格納されているかどうかを示す値のみを返します。データ変数は更新されません。

例: FIND メソッドを使ってハッシュオブジェクト内のキーを検索する

この例では、ハッシュオブジェクトを作成します。2 つのデータ値を追加します。FIND メソッドを使用してハッシュオブジェクト内のキーを検索します。データ値は、キーに関連付けられたデータセット変数に返されます。

```
proc fcmp;
  length k $8;
  length d $12;
  /* Declare hash object and key and data variable names */
```

```
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  /* Define constant key and data values and add to hash object */
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  rc = h.add(key: 'Homer', data: 'Odyssey');
  /* Verify that key JOYCE is in hash object and */
  /* return its data value to the data set variable D */
  rc = h.find(key: 'Joyce');
  put d=;
run;
```

上記のステートメントは、次の結果を生成します。

```
d=Ulysses
```

関連項目

- “データの格納と取得”

メソッド:

- “CHECK メソッド” (162 ページ)
- “DEFINEKEY メソッド” (169 ページ)

FIND_NEXT メソッド

現在のキーの複数アイテムリストで現在のリストアイテムを次のアイテムに設定し、対応するデータ変数にデータを設定します。

適用対象: PROC FCMP ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**FIND_NEXT**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

詳細

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_NEXT メソッドは、キーに複数のデータアイテムが関連付けられているかどうかを判断します。キーに別のデータアイテムがあることが判明した場合、FIND_NEXT メソッドを使用してそのデータアイテムを 取得できます。これによりデータ変数にデータアイテムの値が設定され、メソッド呼び出し後に使用できるようになります。データアイテムリスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドを使用することでリスト内を移動できます。

例

この例では、FIND_NEXT メソッドを使用して、複数のキーに複数のデータアイテムがあるデータセットを反復処理します。キーに複数のデータアイテムがある 場合、後続のアイテムは **dup** とマークされます。

```
data dup;
  length key_id value 8;
  input key_id value;
  datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
  ;
proc fcmp;
file log;
  dcl hash h(dataset:'dup', multidata: 'y');
  rc= h.definekey('key_id');
  rc= h.definedata('key_id', 'value');
  rc= h.definedone();
  /* avoid uninitialized variable notes */
  call missing (key_id, value);
  do key_id = 1 to 5;
    rc = h.find();
    if (rc = 0) then do;
      put @5 key_id= @15 value=;
      rc = h.find_next();
      do while(rc = 0);
```

```
        put 'dup ' @5 key_id= @15 value=;
        rc = h.find_next();
      end;
    end;
  end;
run;
```

次の行が SAS ログに書き出されます。

```
   key_id=1  value=10
dup key_id=1  value=15
dup key_id=1  value=5
   key_id=2  value=11
dup key_id=2  value=16
dup key_id=2  value=9
   key_id=3  value=20
dup key_id=3  value=100
   key_id=4  value=6
   key_id=5  value=5
dup key_id=5  value=99
```

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “FIND メソッド”
- “FIND_PREV メソッド”
- “HAS_NEXT メソッド”

FIND_PREV メソッド

現在のキーの複数アイテムリストで現在のリストアイテムを次のアイテムに設定し、対応するデータ変数にデータを設定します。

適用対象: PROC FCMP ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**FIND_PREV**( );

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

詳細

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_PREV メソッドは、キーに複数のデータアイテムが関連付けられているかどうかを判断します。キーに前のデータアイテムがあることが判明した場合、FIND_PREV メソッドを使用してそのデータアイテムを 取得できます。これによりデータ変数にデータアイテムの値が設定され、メソッド呼び出し後に使用できるようになります。データアイテムリスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドに加えて HAS_PREV メソッドと FIND_PREV メソッドを使用することで、リスト内を移動できます。 例については、“HAS_NEXT メソッド” (61 ページ)を参照してください。

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “FIND メソッド” (53 ページ)
- “FIND_NEXT メソッド” (55 ページ)
- “HAS_PREV メソッド” (63 ページ)

HAS_NEXT メソッド

現在のキーの複数データアイテムリスト内に次のアイテムがあるかどうかを判断します。

適用対象: PROC FCMP ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**HAS_NEXT** (RESULT: R);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

RESULT:R
数値変数 **R** を指定します。この変数は、データアイテムリスト内に別のデータアイテムがない場合はゼロ値、データアイテムリスト内に別のデータアイテムがある場合はゼロ以外の値を受け取ります。

詳細

1 つのキーに複数のデータアイテムがある場合、HAS_NEXT メソッドを使用して、現在のキーの複数データアイテムリスト内に次のアイテムがあるかどうかを判断できます。別のアイテムがある場合、このメソッドは数値変数 **R** でゼロ以外の値を返します。別のアイテムがない場合、ゼロを返します。

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_NEXT メソッドは、キーに複数のデータアイテムが関連付けられているかどうかを判断します。キーに別のデータアイテムがあることが判明した場合、FIND_NEXT メソッドを使用してそのデータアイテムを 取得できます。これによりデータ変数にデータアイテムの値が設定され、メソッド呼び出し後に使用できるようになります。データアイテムリスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドに加えて HAS_PREV メソッドと FIND_PREV メソッドを使用することで、リスト内を移動できます。

例: データアイテムの検索

この例では、複数のキーに複数のデータアイテムがあるハッシュオブジェクトを作成します。さらに、HAS_NEXT メソッドを使用してすべてのデータアイテムを 検索します。

```
data billing;
  length customer_num invoice_amount  8;
  input customer_num invoice_amount;
  datalines;
  1 100
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
proc fcmp;
file log;
  length r 8;
```

```
  dcl hash h(dataset:'billing', multidata: 'y');
  rc= h.definekey('customer_num');
  rc= h.definedata('customer_num', 'invoice_amount');
  rc= h.definedone();
  call missing (customer_num, invoice_amount);
  do customer_num = 1 to 5;
    rc = h.find();
    if (rc = 0) then do;
      put @5 customer_num= @20 invoice_amount=;
      rc= h.has_next(result: r);
      do while(r ne 0);
        rc = h.find_next();
        put 'dup ' @5 customer_num= @20 invoice_amount=;
        rc= h.has_next(result: r);
      end;
    end;
  end;
run;
```

次の行が SAS ログに書き出されます。

```
   customer_num=1 invoice_amount=100
dup customer_num=1 invoice_amount=15
dup customer_num=1 invoice_amount=5
   customer_num=2 invoice_amount=11
dup customer_num=2 invoice_amount=16
dup customer_num=2 invoice_amount=9
   customer_num=3 invoice_amount=20
dup customer_num=3 invoice_amount=100
   customer_num=4 invoice_amount=6
   customer_num=5 invoice_amount=5
dup customer_num=5 invoice_amount=99
```

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “FIND メソッド”
- “FIND_NEXT メソッド”
- “FIND_PREV メソッド”
- “HAS_NEXT メソッド”

HAS_PREV メソッド

現在のキーの複数データアイテムリスト内に前のアイテムがあるかどうかを判断します。

適用対象: ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**HAS_PREV** (RESULT: R);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

RESULT:R
数値変数 **R** を指定します。この変数は、データアイテムリスト内に別のデータアイテムがない場合はゼロ値、データアイテムリスト内に別のデータアイテムがある場合はゼロ以外の値を受け取ります。

詳細

1 つのキーに複数のデータアイテムがある場合、HAS_PREV メソッドを使用して、現在のキーの複数データアイテムリスト内に前のアイテムがあるかどうかを判断できます。前のアイテムがある場合、このメソッドは数値変数 **R** でゼロ以外の値を返します。別のアイテムがない場合、ゼロを返します。

FIND メソッドは、キーがハッシュオブジェクト内に存在するかどうかを判断します。HAS_NEXT メソッドは、キーに複数のデータアイテムが関連付けられているかどうかを判断します。キーに前のデータアイテムがあることが判明した場合、FIND_PREV メソッドを使用してそのデータアイテムを 取得できます。これによりデータ変数にデータアイテムの値が設定され、メソッド呼び出し後に使用できるようになります。データアイテムリスト内では、HAS_NEXT メソッドと FIND_NEXT メソッドに加えて HAS_PREV メソッドと FIND_PREV メソッドを使用することで、リスト内を移動できます。 例については、“HAS_NEXT メソッド” (61 ページ)を参照してください。

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “FIND メソッド”
- “FIND_NEXT メソッド”
- “FIND_PREV メソッド”

■ “HAS_NEXT メソッド”

DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト

PROC FCMP ハッシュオブジェクトまたは PROC FCMP ハッシュ反復子オブジェクトを宣言し、PROC FCMP ハッシュオブジェクトまたは PROC FCMP ハッシュ反復子オブジェクトのインスタンスを作成し、データを初期化します。

該当要素: PROC FCMP

カテゴリ: アクション

種類: 宣言

別名: DCL

適用対象: PROC FCMP ハッシュオブジェクト, PROC FCMP ハッシュ反復子オブジェクト

構文

形式 1: **DECLARE HASH** *object-reference*;

形式 2: **DECLARE HITER** *object-reference* ("*hash-reference*");

形式 3: **DECLARE** *object object-reference* (<*argument_tag-1*: *value-1*, *argument_tag-2*: *value-2*, ...>);

引数

object
コンポーネントオブジェクトを指定します。*object* には次のいずれかの値を指定できます。

hash
ハッシュオブジェクトを指定します。ハッシュオブジェクトは、迅速なデータの保存および取得のメカニズムを提供します。ハッシュオブジェクトは、ルックアップキーに基づいてデータを保存および取得します。

参照項目: 2 章, “ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用”

hiter
ハッシュ反復子オブジェクトを指定します。ハッシュ反復子オブジェクトを使用すると、ハッシュオブジェクトのデータを正または逆のキー順序で取得できます。

参照項目: 2 章, “ハッシュオブジェクトおよびハッシュ反復子オブジェクトの使用”

object-reference
ハッシュオブジェクトまたはハッシュ反復子オブジェクトのオブジェクト参照名を指定します。

hash-reference
ハッシュ反復子を添付するハッシュオブジェクトの参照名を指定します。

argument_tag: *value*
ハッシュオブジェクトのインスタンスの作成に使用する情報を指定します。

有効なハッシュオブジェクト引数および値タグには、次の 4 つがあります。

dataset: '*dataset_name* <(*datasetoption*)>'
ハッシュオブジェクトに読み込む SAS データセットの名前を指定します。

SAS データセットの名前はリテラルでなければなりません。リテラルデータセット名は単一引用符または二重引用符で囲む必要があります。

この構文が使用されます。

```
dcl hash h (dataset: 'x');
```

注 データセットに重複するキーが含まれている場合、デフォルトでは、ハッシュオブジェクトの最初のインスタンスがロードされ、後続のインスタンスは無視されます。DUPLICATE 引数タグを使用して、最後のインスタンスをハッシュオブジェクトに格納するか、SAS ログにエラーメッセージを書き込みます。

duplicate: '*option*'
重複したキーが許されていない場合の処理方法を決定します。デフォルトでは、最初のキーが保存され、後続の重複は無視されます。*option* には次のいずれかの値を指定できます。

'replace'
'r'
最後の重複するキーレコードを保存します。

'error'
'e'
重複するキーが検出された場合に、ログにエラーを報告します。

REPLACE オプションを使用するこの例では、 *color_id* キー 531 には **blue**、*color_id* キー 620 には **brown** が保存されます。デフォルトでは、最初の値 (531 は **yellow**、620 は **green**)がロードされ、重複する値は無視されます。

```
data table;
 input color_id color_name $;
 datalines;
 531 yellow
 620 green
 531 blue
 908 orange
 620 brown
 143 purple
run;

proc fcmp;
length color_id 8 color_name $ 8;
if (_n_ = 1) then do;
```

```
    declare hash myhash(dataset: "table", duplicate: "r");
    rc = myhash.definekey('color_id');
    rc = myhash.definedata('color_id','color_name');
    rc=myhash.definedone();
    /* avoid uninitialized variable notes */
    call missing(color_id, color_name);
  end;
  n = myhash.num_items();
  put n=;
 run;
```

```
n=4
```

hashexp: *n*

ハッシュオブジェクトの内部テーブルサイズです。ハッシュテーブルのサイズは 2^n です。

HASHEXP の値は、ハッシュテーブルサイズを作成する 2 の累乗指数として使用されます。たとえば、HASHEXP の値 4 は、ハッシュテーブルサイズ 2^4(つまり 16)と同じです。HASHEXP の最大値は 20 です。

ハッシュテーブルサイズは、保存可能なアイテム数とは等しくありません。ハッシュテーブルを、一連の'バケット'と考えます。ハッシュテーブルサイズ 16 には、16 個のバケットがあります。各バケットに保持できるアイテムの数は無限です。ハッシュテーブルの効率は、アイテムをバケットにマップし、バケットからアイテムを取得するハッシュ関数の機能にあります。

ハッシュオブジェクトのルックアップルーチンの効率を最大にするには、ハッシュオブジェクトのデータ量に応じてハッシュテーブルサイズを指定する必要があります。最適な結果が得られるまで、さまざまな HASHEXP 値を試してみてください。例えば、ハッシュオブジェクトに 100 万個のアイテムが含まれている場合、ハッシュテーブルサイズ 16 (HASHEXP=4)でも機能しますが、効率はあまり良くありません。ハッシュテーブルサイズ 512 または 1024 (HASHEXP=9 または 10)を使用すると、最高の処理速度が得られます。

デフォルト　8。これは、ハッシュテーブルサイズ 2^8(つまり 256)と同じです。

multidata: '*option*'

デフォルト　NO

制限事項　MULTIDATA 引数は、2024.11 のリリースから利用可能になります。

ヒント　引数値は二重引用符で囲むこともできます。

各キーに複数のデータアイテムを許可するかどうかを指定します。

option には、次のいずれかの値を使用できます。

'YES'
'Y'
　各キーに複数のデータアイテムが許可されます。

'NO'
'N'
　各キーにデータアイテムが 1 つのみ許可されます。

ordered: '*option*'
ハッシュオブジェクトとハッシュ反復子オブジェクトを併用する場合、データがキー値の順序で返されるかどうか、またはどのように返されるかを指定します。

option には次のいずれかの値を指定できます。

'ascending'
'a'
データはキー値の昇順で返されます。'**ascending**'の指定は、'**yes**'を指定することと同じです。

'descending'
'd'
データはキー値の降順で返されます。

'YES'
'Y'
データはキー値の昇順で返されます。'**yes**'の指定は、'**ascending**'を指定することと同じです。

'NO'
'N'
データは未定義の順序で返されます。

デフォルト NO

ヒント 引数は二重引用符で囲むこともできます。

suminc: '*variable-name*'
制限事項 SUMINC 引数は、2024.11 のリリースから利用可能になります。

参照項目: “キー集計の維持”

例 キー集計は、変数の現在の値を使用して変更されます。
```
dcl hash myhash(suminc: 'count');
```

ハッシュオブジェクトキーの集計数を維持します。SUMINC 引数タグは、PROC FCMP 変数を指定します。この変数には、合計増分が保持されます。合計増分はキーが参照されるたびにキー集計に追加される数です。

参照項目: “コンストラクターを使ったハッシュオブジェクトデータの初期化”

詳細

基本情報

SAS プログラムで PROC FCMP コンポーネントオブジェクトを使用するには、オブジェクトを宣言して作成(インスタンス化)する必要があります。定義済みのハッシュおよびハッシュ反復子コンポーネントオブジェクトには、PROC FCMP コンポーネントインターフェイスからアクセスできます。

ハッシュオブジェクトの宣言(フォーム 1)

DECLARE ステートメントを使用してハッシュオブジェクトを宣言します。

```
declare hash h;
```

DECLARE ステートメントは、オブジェクト参照 H がハッシュオブジェクトであることを SAS に示します。

DECLARE ステートメントを使用したハッシュオブジェクトまたはハッシュ反復子オブジェクトのインスタンス生成(フォーム 2)

DECLARE ステートメントを使用してハッシュオブジェクトまたはハッシュ反復子オブジェクトの宣言とインスタンス化を 1 つのステップで実行します。例えば、次のコード行では、DECLARE ステートメントでハッシュオブジェクトが宣言およびインスタンス化され、オブジェクト参照 H に割り当てられます。

```
declare hash h( );
```

constructor は、ハッシュオブジェクトをインスタンス化し、ハッシュオブジェクトデータを初期化するために使用できるメソッドです。例えば、次のコード行では、DECLARE ステートメントでハッシュオブジェクトが宣言およびインスタンス化され、オブジェクト参照 H に割り当てられます。さらに、引数タグ HASHEXP を使用してハッシュテーブルサイズが値 16(2^4)に初期化されます。

```
declare hash h(hashexp: 4);
```

例

例 1: DECLARE ステートメントを使用したハッシュオブジェクトの宣言とインスタンス生成

この例では、DECLARE ステートメントを使用してハッシュオブジェクトの宣言とインスタンス化を 1 つのステップで実行します。

```
proc fcmp;
  length k $15;
  length d $15;
  if _N_ = 1 then do;
    /* Declare and instantiate hash object "myhash" */
    declare hash myhash( );
    rc = myhash.defineKey('k');
    rc = myhash.defineData('d');
    rc = myhash.defineDone( );
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;

  /* Create constant key and data values */
  rc = myhash.add(key: 'Labrador', data: 'Retriever');
  rc = myhash.add(key: 'Airedale', data: 'Terrier');
  rc = myhash.add(key: 'Standard', data: 'Poodle');
  /* Find data associated with key and write data to log*/
  rc = myhash.find(key: 'Airedale');
```

```
  if (rc = 0) then
    put d=;
  else
    put 'Key Airedale not found';
run;
```

上記のステートメントは、次の結果を生成します。

```
d=Terrier
```

例 2: ハッシュオブジェクトのインスタンスを生成してサイズ順に並べる

この例では、DECLARE ステートメントを使用してハッシュオブジェクトを宣言し、インスタンス化します。ハッシュテーブルサイズは 16(2^4)に設定されています。

```
proc fcmp;
  length k $15;
  length d $15;
  if _N_ = 1 then do;
    /* Declare and instantiate hash object "myhash". */
    /* Set hash table size to 16. */
    declare hash myhash(hashexp: 4);
    rc = myhash.defineKey('k');
    rc = myhash.defineData('d');
    rc = myhash.defineDone( );
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;

  /* Create constant key and data values */
  rc = myhash.add(key: 'Labrador', data: 'Retriever');
  rc = myhash.add(key: 'Airedale', data: 'Terrier');
  rc = myhash.add(key: 'Standard', data: 'Poodle');
  rc = myhash.find(key: 'Airedale');
  /* Find data associated with key and write data to log*/
  if (rc = 0) then
    put d=;
  else
    put 'Key Airedale not found';
run;
```

上記のステートメントは、次の結果を生成します。

```
d=Terrier
```

NUM_ITEMS メソッド 属性

PROC FCMP ハッシュオブジェクト内のアイテム数を返します。

適用対象: PROC FCMP ハッシュオブジェクト

構文

variable_name=*object*.**NUM_ITEMS**();

引数

variable_name
ハッシュオブジェクト内のアイテム数を含む変数名を指定します。

object
ハッシュオブジェクト名を指定します。

例: ハッシュオブジェクト内のアイテム数を返す

この例では、データセットを作成し、そのデータセットをハッシュオブジェクト内に読み込みます。アイテムがハッシュオブジェクトに追加され、その結果のハッシュオブジェクト内のアイテムの合計数が NUM_ITEMS METHOD によって返されます。

```
data work.stock;
  input item $ qty;
  datalines;
broccoli 345
corn 389
potato 993
onion 730
;
proc fcmp;
  if _N_ = 1 then do;
    length item $10;
    length qty 8;
    length totalitems 8;
    /* Declare hash object and read STOCK data set as ordered */
    declare hash myhash(dataset: "work.stock");
    /* Define key and data variables */
    rc=myhash.defineKey('item');
    rc=myhash.defineData('qty');
    rc=myhash.defineDone();
  end;
  /* Add a key and data value to the hash object */
  item = 'celery';
  qty = 183;
  rc = myhash.add();
  if (rc ne 0) then
    put 'Add failed';
  /* Use NUM_ITEMS to return updated number of items in hash object */
  totalitems = myhash.num_items();
  put totalitems=;
run;
```

上記のステートメントは、次の結果を生成します。

```
totalitems=5
```

REF メソッド

CHECK メソッドと ADD メソッドを 1 つのメソッド呼び出しに統合します。

適用対象: ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**REF** (<<KEY: *keyvalue-1*>, ...<KEY: *keyvalue-n*>, <DATA: *datavalue-1*> , ...<DATA: *datavalue-n*>>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

“KEY: *keyvalue*”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 *keyvalue* にコンポーネントオブジェクトを含めることはできません。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

“DATA: *datavalue*”ペアの数は、DEFINEDATA メソッドを使用して定義されたデータ変数の数によって変わります。

制限事項 *datavalue* にコンポーネントオブジェクトを含めることはできません。

詳細

CHECK メソッドと ADD メソッドを 1 つの REF メソッド呼び出しに統合できます。次のコードを変更できます。

```
rc = h.check();
 if (rc ne 0) then
   rc = h.add();
```

to

```
rc = h.ref();
```

REF メソッドは、ハッシュオブジェクト内の各キーの出現数を数えるのに便利です。REF メソッドは、最初の ADD で各キーのキー集計を初期化し、後続の CHECK が出現するたびに ADD を変更します。

キー集計の詳細については、“キー集計の維持”を参照してください。

例: REF メソッドを使用したキー集計

次の例では、キー集計に REF メソッドを使用します。

```
data keys;
 input key_id;
datalines;
1
2
1
3
5
2
3
2
4
1
5
1
;

proc fcmp data=keys;
file log;
 length count key_id 8;
 if _n_ = 1 then do;
  declare hash myhash(suminc: "count", ordered: "y");
  declare hiter iter("myhash");
  rc = myhash.defineKey('key_id');
  rc = myhash.defineDone();
  count = 1;
 end;

 rc = myhash.ref();
 file log;
 if (_n_ = 12) then do;
    rc = iter.first();
```

```
    do while(rc = 0);
    rc = myhash.sum(sum: count);
      put key_id= count=;
      rc = iter.next();
    end;
 end;
run;
```

次の行が SAS ログに書き出されます。

```
key_id=1 count=4
key_id=2 count=3
key_id=3 count=2
key_id=4 count=1
key_id=5 count=2
```

関連項目

メソッド:

- “ADD メソッド”
- “CHECK メソッド”

REMOVEDUP メソッド

指定したキーの現在のデータアイテムに関連付けられているデータをハッシュオブジェクトから削除します。

適用対象: PROC FCMP ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**REMOVEDUP** (<KEY: *keyvalue-1*, ...KEY: *keyvalue-n*>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

“KEY: *keyvalue*”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 関連付けられているハッシュ反復子が *keyvalue* を示している場合、REMOVEDUP メソッドはハッシュオブジェクトからキーまたはデータを削除しません。エラーメッセージが発行されます。

詳細

REMOVEDUP メソッドは、ハッシュオブジェクトからキーとデータの両方を削除します。

次の 2 つの方法のいずれかで REMOVEDUP メソッドを使用して、ハッシュオブジェクトのキーおよびデータを削除できます。1 つ目は、キーを指定してから、REMOVEDUP メソッドを使用する方法です。2 つ目は、ショートカットを使用して、REMOVEDUP メソッド呼び出しでキーを直接指定する方法です。

注: REMOVEDUP メソッドでは、データ変数の値は変更されません。ハッシュオブジェクトの値のみが削除されます。

注: キーのデータアイテムリストにあるデータアイテムが 1 つのみの場合は、キーとデータがハッシュオブジェクトから削除されます。

比較

REMOVEDUP メソッドでは、指定したキーの現在のデータアイテムに関連付けられているデータがハッシュオブジェクトから削除されます。REMOVE メソッドでは、指定したキーに関連付けられているデータがハッシュオブジェクトから削除されます。

例: キーの重複するアイテムの削除

この例では、複数のキーに複数のデータアイテムがあるハッシュオブジェクトを作成します。キーの 2 番目のデータアイテムが削除されます。

```
data testdup;
  length key_id value 8;
  input key_id value;
  datalines;
  1 10
  2 11
```

```
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
proc fcmp;
file log;
  length r 8;
  dcl hash h(dataset:'testdup', multidata: 'y', ordered: 'y');
  rc= h.definekey('key_id');
  rc= h.definedata('key_id', 'value');
  rc= h.definedone();
  call missing (key_id, value);
  do key_id = 1 to 5;
    rc = h.find();
    if (rc = 0) then do;
      rc= h.has_next(result: r);
      if (r ne 0) then do;
        rc= h.find_next();
        rc= h.removedup();
      end;
    end;
  end;
  dcl hiter i('h');
  rc = i.first();
  do while (rc = 0);
    put key_id= value=;
    rc = i.next();
  end;
run;
```

次の行が SAS ログに書き出されます。

```
key_id=1 value=10
key_id=1 value=5
key_id=2 value=11
key_id=2 value=9
key_id=3 value=20
key_id=4 value=6
key_id=5 value=5
```

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “REMOVE メソッド”

REMOVE メソッド

指定したキーに関連付けられているデータを PROC FCMP ハッシュオブジェクトから削除します。

適用対象: PROC FCMP ハッシュオブジェクト

構文

rc=*object*.**REMOVE** (<KEY: *keyvalue-1*, KEY: *keyvalue-2*, ...>);

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。*keyvalue* はリテラルでなければなりません。

“KEY: *keyvalue*”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

制限事項 関連付けられているハッシュ反復子が *keyvalue* を参照している場合、REMOVE メソッドはハッシュオブジェクトからキーまたはデータを削除しません。

詳細

REMOVE メソッドは、ハッシュオブジェクトからキーとデータの両方を削除します。

次の 2 つの方法のいずれかで REMOVE メソッドを使用して、ハッシュオブジェクトのキーおよびデータを削除できます。

1 つ目は、このコードに示すように、キーを指定してから REMOVE メソッドを使用する方法です。

```
proc fcmp;
  length k $8;
  length d $12;
  if _N_ = 1 then do;
    declare hash h();
```

```
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  /* Specify the key */
  k = 'Joyce';
  /* Use the REMOVE method to remove the key and data */
  rc = h.remove();
  if (rc = 0) then
    put 'Key and data removed from the hash object.';
run;
```

2 つ目は、このコードに示すように、ショートカットを使用して、REMOVE メソッド呼び出しでキーを直接指定する方法です。

```
proc fcmp;
  length k $8;
  length d $12;
  if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k');
    rc = h.defineData('d');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(k, d);
  end;
  rc = h.add(key: 'Joyce', data: 'Ulysses');
  rc = h.add(key: 'Homer', data: 'Iliad');
  /* Specify the key in the REMOVE method parameter */
  rc = h.remove(key: 'Homer');
  if (rc  =0) then
    put 'Key and data removed from the hash object.';
run;
```

注: REMOVE メソッドではデータ変数の値は変更されません。メソッドはハッシュオブジェクトの値のみを削除します。

関連項目

メソッド:

- “ADD メソッド” (160 ページ)
- “DEFINEKEY メソッド” (169 ページ)

REPLACE メソッド

指定したキーに関連付けられているデータを PROC FCMP ハッシュオブジェクトの新しいデータと置き換えます。

適用対象: PROC FCMP ハッシュオブジェクト

構文

rc=*object*.**REPLACE** (<<KEY: *keyvalue-1*>, <KEY: *keyvalue-2*, ...>, <DATA: *datavalue-1*>,
<DATA: *datavalue-2*>, ...>);

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、対応するエラーメッセージがログに書き込まれます。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。*keyvalue* はリテラルでなければなりません。

“KEY: *keyvalue*”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

要件 KEY: *keyvalue* 引数は、ハッシュオブジェクト変数名が指定されていないため、ハッシュオブジェクトで定義された順序になっていなければなりません。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。*datavalue* はリテラルでなければなりません。

“DATA: *datavalue*”ペアの数は、DEFINEDATA メソッドを使用して定義されたデータ変数の数によって変わります。

要件 DATA: *datavalue* 引数は、ハッシュオブジェクト変数名が指定されていないため、ハッシュオブジェクトで定義された順序になっていなければなりません。

詳細

次の 2 つの方法のいずれかで REPLACE メソッドを使用して、ハッシュオブジェクト内のデータを置き換えることができます。

1 つ目は、次のコードに示すように、キーおよびデータアイテムを定義してから REPLACE メソッドを使用してデータを置き換える方法です。この例では、キー 'Collie'の **Place** データが'BestInShow'から'bis'に変更されます。

> ヒント　キーは defineData メソッドでデータアイテムとして定義され、データセットに書き込まれるようになっています。

```
data work.dogshow;
  length breed $10 place $10;
  input breed place;
datalines;
Terrier    2nd
Beagle     3rd
Rottweiler 1st
Collie     BestInShow
Poodle     2nd
Boxer      3rd
;

proc fcmp;
  length breed $10 place $10;
  if _N_ = 1 then do;
    declare hash h(dataset: 'work.dogshow');
    rc = h.defineKey('breed');
    rc = h.defineData('breed', 'place');
    rc = h.defineDone();
  end;
  /* Specify the key and new data value */
  breed = 'Collie';
  place = 'bis';
  /* Call the REPLACE method to replace the data value */
  rc = h.replace();

run;
```

2 つ目は、次のコードに示すように、ショートカットを使用して、REPLACE メソッド呼び出しでキーおよびデータを直接指定する方法です。

注: キーは DEFINEDATA メソッドでデータアイテムとして定義されているので、REPLACE メソッドではキーとデータアイテムの両方を含める必要があります。

```
data work.dogshow2;
  length breed $10 place $10;
  input breed place;
datalines;
Terrier    2nd
Beagle     3rd
Rottweiler 1st
```

```
Collie    BestInShow
Poodle    2nd
Boxer     3rd
;

proc fcmp;
  length breed $10 place $10;
  if _N_ = 1 then do;
    declare hash h(dataset:'work.dogshow2');
    rc = h.defineKey('breed');
    rc = h.defineData('breed', 'place');
    rc = h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(breed, place);
  end;
  /* Specify the key and new data value in the REPLACE method */
  rc = h.replace(key: 'Collie', data: 'Collie', data: 'bis');
run;
```

注: REPLACE メソッドを呼び出してキーが見つからなかった場合、キーとデータがハッシュオブジェクトに追加されます。

注: REPLACE メソッドでは、データ変数の値はデータアイテムの値に置き換えられません。メソッドはハッシュオブジェクトの値のみを置き換えます。

関連項目

メソッド:

- “DEFINEDATA メソッド” (165 ページ)
- “DEFINEKEY メソッド” (169 ページ)

REPLACEDUP メソッド

現在のキーの現在のデータアイテムに関連付けられたデータを新しいデータに置き換えます。

適用対象: PROC FCMP ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**REPLACEDUP** (<DATA: *datavalue-1*, ...DATA: *datavalue-n*>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

DATA: *datavalue*
DEFINEDATA メソッド呼び出しで指定された、対応するデータ変数に一致する型のデータ値を指定します。

“DATA: *datavalue*”ペアの数は、DEFINEDATA メソッドを使用して現在のキーに定義されたデータ変数の数によって変わります。

詳細

次の 2 つの方法のいずれかで REPLACEUP メソッドを使用して、ハッシュオブジェクト内のデータを置き換えることができます。

1 つ目は、データアイテムを定義してから、REPLACEDUP メソッドを使用する方法です。2 つ目は、ショートカットを使用して、REPLACEDUP メソッド呼び出しでデータを直接指定する方法です。

注: REPLACEDUP メソッドを呼び出してキーが見つからなかった場合、キーとデータがハッシュオブジェクトに追加されます。

注: REPLACEDUP メソッドでは、データ変数の値はデータアイテムの値に置き換えられません。ハッシュオブジェクト内の値のみが置換されます。

比較

REPLACEDUP メソッドでは、現在のキーの現在のデータアイテムに関連付けられたデータが新しいデータに置き換えられます。REPLACE メソッドでは、指定したキーに関連付けられたデータが新しいデータに置き換えられます。

例: 現在のキーのデータの置換

この例では、複数のキーに複数のデータアイテムがあるハッシュオブジェクトを作成します。重複データアイテムが見つかった場合、データアイテムの値に 300 が加算されます。

```
data testdup;
  length key_id value 8;
  input key_id value;
```

```
  datalines;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
proc fcmp;
file log;
  length r 8;
  dcl hash h(dataset:'testdup', multidata: 'y', ordered: 'y');
  rc= h.definekey('key_id');
  rc= h.definedata('key_id', 'value');
  rc= h.definedone();
  call missing (key_id, value);
  do key_id = 1 to 5;
    rc = h.find();
    if (rc = 0) then do;
      put @5 key_id= @15 value=;
      rc= h.has_next(result: r);
      do while(r ne 0);
        rc = h.find_next();
        put 'dup ' @5 key_id= @15 value=;
        value = value + 300;
        rc = h.replacedup();
        rc= h.has_next(result: r);
      end;
    end;
  end;
  put 'iterating...';
  dcl hiter i('h');
  rc = i.first();
  do while (rc = 0);
    put @5 key_id= @15 value=;
    rc = i.next();
   end;
run;
```

次の行が SAS ログに書き出されます。

```
   key_id=1  value=10
dup key_id=1  value=15
dup key_id=1  value=5
   key_id=2  value=11
dup key_id=2  value=16
dup key_id=2  value=9
   key_id=3  value=20
dup key_id=3  value=100
   key_id=4  value=6
   key_id=5  value=5
dup key_id=5  value=99
iterating...
   key_id=1  value=10
   key_id=1  value=315
   key_id=1  value=305
   key_id=2  value=11
   key_id=2  value=316
   key_id=2  value=309
   key_id=3  value=20
   key_id=3  value=400
   key_id=4  value=6
   key_id=5  value=5
   key_id=5  value=399
```

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “REPLACE メソッド”

RESET_DUP メソッド

DO_OVER メソッド使用時にポインターをキーの重複リストの最初にリセットします。

適用対象: PROC FCMP ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**RESET_DUP**( );

引数

rc

メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

詳細

ハッシュオブジェクトが単一のキーに対して複数の値を持つ場合、DO_OVER メソッドを DO ループの反復内で使用して、重複キー間を移動します。DO_OVER メソッドは最初のメソッド呼び出しでキーを読み込み、キーが最後に到達するまで重複キーリストを移動しつづけます。

反復の途中でキーを切り替えた場合、RESET_DUP メソッドを使用してポインターをリストの最初にリセットしなければなりません。そうしないと、SAS は最初のキーを使い続けます。

サンプルは DO_OVER メソッド例 (49 ページ)を参照してください。

関連項目

メソッド:

- “DO_OVER メソッド”

SUMDUP メソッド

現在のキーの現在のデータアイテムについて 集計値を取得し、その値を DATA ステップ変数に保存します。

適用対象: PROC FCMP ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**SUMDUP** (SUM: *variable-name*);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

SUM: *variable-name*
現在のキーの現在のデータアイテムについて 取得した集計値を保存する DATA ステップ変数を指定します。

詳細

キーに複数のデータアイテムがある場合、ハッシュオブジェクトからキーの集計値を取得するには、SUMDUP メソッドを使用します。詳細については、“キー集計の維持”を参照してください。

比較

SUMDUP メソッドでは、1 つのキーにつき複数のデータアイテムが存在するときに、現在のキーの現在のデータアイテムについて 集計値を取得します。SUM メソッドでは、1 つのキーにつき 1 つのデータアイテムしか存在しないときに、特定のキーの集計値を取得します。

例: 集計値の取得

次の例では、SUMDUP メソッドを使って、現在のデータアイテムの集計値を取得します。このメソッドは、HAS_PREV メソッドと FIND_PREV メソッドを使うと、リストの前方にループできることも示しています。FIND_PREV メソッドは、現在のリストアイテムについては FIND_NEXT メソッドと同様の処理を行います。異なるのは、複数のアイテムリストで前方に移動する点です。

```
data dup;
  length key_id value 8;
  input key_id value;
  cards;
  1 10
  2 11
  1 15
  3 20
  2 16
  2 9
  3 100
  5 5
  1 5
  4 6
  5 99
;
proc fcmp;
file log;
  length r i sum 8;
  i = 0;
  dcl hash h(dataset:'dup', multidata: 'y', suminc: 'i');
  rc= h.definekey('key_id');
  rc= h.definedata('key_id', 'value');
```

```
 rc= h.definedone();
 call missing (key_id, value);
 i = 1;
do key_id = 1 to 5;
  rc = h.find();
  if (rc = 0) then do;
    rc= h.has_next(result: r);
    do while(r ne 0);
      rc = h.find_next();
      rc = h.find_prev();
      rc = h.find_next();
      rc= h.has_next(result: r);
    end;
  end;
end;
i = 0;
do key_id = 1 to 5;
 rc = h.find();
 if (rc = 0) then do;
    rc= h.sum(sum: sum);
    put @5 key_id= @15 value= @26 sum=;
    rc= h.has_next(result: r);
    do while(r ne 0);
      rc = h.find_next();
      rc= h.sumdup(sum: sum);
      put 'dup ' @5 key_id= @15 value= @26 sum=;
      rc= h.has_next(result: r);
    end;
  end;
end;
run;
```

次の行が SAS ログに書き出されます。

```
   key_id=1  value=10   sum=2
dup key_id=1  value=15   sum=3
dup key_id=1  value=5    sum=2
   key_id=2  value=11   sum=2
dup key_id=2  value=16   sum=3
dup key_id=2  value=9    sum=2
   key_id=3  value=20   sum=2
dup key_id=3  value=100  sum=2
   key_id=4  value=6    sum=1
   key_id=5  value=5    sum=2
dup key_id=5  value=99   sum=2
```

このメソッドがどのように機能するかを確認するために、3 つの値 10、15、5 (この順番で格納)を持つ key_id 1 を考えてみましょう。

```
   key_id=1  value=10   sum=2
dup key_id=1  value=15   sum=3
dup key_id=1  value=5    sum=2
```

最初の do key_id = 1 to 5;ループの値リストを移動中、最初の FIND メソッド呼び出しで値アイテム 10 のキー集計値が 1 に設定されます。最初の FIND_NEXT メソッド呼び出しで値アイテム 15 のキー集計値が 1 に設定されます。次の FIND_PREV メソッド呼び出しでは、値アイテム 10 に戻り、キー集計値が 2 に増分されます。さらに、最後の FIND_NEXT メソッド呼び出しで、値アイテム 15 のキー集計値が 2 に設

定されます。ループの次の反復では、値アイテム 5 のキー集計値が 1 に設定され、値アイテム 15 のキー集計値が 3 に設定されます。最後に、値アイテム 5 のキー集計値が 2 に増分されます。

この例では、リストに以前のエントリが存在していることがわかっているため、FIND_PREV メソッドを呼び出す前に HAS_PREV メソッドを呼び出していません。存在しなかった場合は、ループに入りません。

このことから、重複処理によっては特殊なメソッドが必要なこともわかります。(この場合、現在の値アイテムのキー集計値を取得することで、SUMDUP メソッドは SUM メソッドと同様の処理を行います。)

関連項目

- “キー値とデータのペアの重複”

メソッド:

- “SUM メソッド”

SUM メソッド

ハッシュテーブルから特定のキーの集計値を取得し、その値を DATA ステップ変数に保存します。

適用対象: PROC FCMP ハッシュオブジェクト

制限事項: このメソッドは、2024.11 のリリースから利用可能になります。

構文

rc=*object*.**SUM** (<KEY: *keyvalue-1*, ...KEY: *keyvalue-n*,>SUM: *variable-name*);

必須引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object
ハッシュオブジェクト名を指定します。

KEY: *keyvalue*
DEFINEKEY メソッド呼び出しで指定された、対応するキー変数に一致する型のキー値を指定します。

“KEY: *keyvalue*”ペアの数は、DEFINEKEY メソッドを使用して定義されたキー変数の数によって変わります。

SUM: *variable-name*
特定のキーの現在の集計値を保存する DATA ステップ変数を指定します。

詳細

ハッシュオブジェクトからキーの集計値を取得するには、SUM メソッドを使用します。詳細については、“キー集計の維持”を参照してください。

比較

SUM メソッドでは、1 つのキーにつき 1 つのデータアイテムしか存在しないときに、特定のキーの集計値を取得します。SUMDUP メソッドでは、1 つのキーにつき複数のデータアイテムが存在するときに、現在のキーの現在のデータアイテムについて集計値を取得します。

例: 特定のキーのキー集計値の取得

次の例では、2 つのキー K=99 および K=100 を指定し、SUM メソッドを使用して各キーの集計値を取得します。

```
proc fcmp;
file log;
  retain total 0;
  if _N_=1 then
    do;
      declare hash h(suminc:'count');
      rc=h.defineKey('k');
      rc=h.definedone();
    end;
  k = 99;
  count = 1;
  rc= h.add();
  /* key=99 summary is now 1 */
  k = 100;
  rc= h.add();
  /* key=100 summary is now 1 */
  k = 99;
  rc= h.find();
  /* key=99 summary is now 2 */
  count = 2;
  rc= h.find();
  /* key=99 summary is now 4 */
  k = 100;
  rc= h.find();
  /* key=100 summary is now 3 */
  rc= h.sum(sum: total);
  put 'total for key 100 = ' total;
  k = 99;
```

```
    rc= h.sum(sum:total);
    put 'total for key 99 = ' total;
run;
```

最初の PUT ステートメントは k=100 の集計値を出力します。

```
total for key 100 = 3
```

2 番目の PUT ステートメントは k=99 の集計値を出力します。

```
total for key 99 = 4
```

関連項目

メソッド:

- “ADD メソッド”
- “FIND メソッド” (176 ページ)
- “CHECK メソッド” (162 ページ)
- “DEFINEKEY メソッド” (169 ページ)
- “REF メソッド”
- “SUMDUP メソッド” (205 ページ)

演算子:

- “_NEW_ 演算子: ハッシュとハッシュ反復子” (67 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト”

FIRST メソッド

基となる PROC FCMP ハッシュオブジェクトの開始値を返します。

適用対象: PROC FCMP ハッシュ反復子オブジェクト

構文

rc=*object*.**FIRST**( );

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しにリターンコード変数を指定せずにメソッドが失敗した場合、キーが見つからないことを示すエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

FIRST メソッドは、ハッシュオブジェクト内の最初のデータアイテムを返します。ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントで **ordered: 'yes'**または **ordered: 'ascending'**引数タグを使用した場合、ハッシュオブジェクト内のデータアイテムはキー値の昇順で並べ替えられるため、'最小'キー(最小の数値または最初のアルファベット文字)のデータアイテムが返されます。NEXT メソッドへの繰り返される呼び出しは、ハッシュオブジェクト内を反復して移動し、キーの昇順でデータアイテムを返します。反対に、ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントで **ordered: 'descending'**引数タグを使用した場合、ハッシュオブジェクト内のデータアイテムはキー値の降順で並べ替えられるため、'最大'キー(最大の数値または最後のアルファベット文字)のデータアイテムが返されます。NEXT メソッドへの繰り返される呼び出しは、ハッシュオブジェクト内を反復して移動し、キーの降順でデータアイテムを返します。

ハッシュオブジェクト内の最後のデータアイテムを返すには、LAST メソッドを使用します。

注: FIRST メソッドは、データ変数にデータアイテムの値を設定し、メソッド呼び出し後にそのデータアイテムを 使用できるようにします。

例: ハッシュオブジェクトデータの取得

この例では、売上データを含むデータセットを作成します。製品を売上順に表示します。データがハッシュオブジェクト内に読み込まれ、データの取得に FIRST メソッドと NEXT メソッドが使用されます。

```
data work.sales;
  input prod $1-6 qty $9-14;
  datalines;
banana  398487
apple   384223
orange  329559
;
proc fcmp;
    length prod $10 qty $6;
  /* Declare hash object and read SALES data set as ordered */
  if _N_ = 1 then do;
    declare hash h(dataset: 'work.sales', ordered: 'yes');
    declare hiter iter('h');
    /* Define key and data variables */
    rc=h.defineKey('qty');
```

```
    rc=h.defineData('prod', 'qty');
    rc=h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(qty, prod);
  end;
  /* Iterate through the hash object and output data values */
  rc = iter.first();
  do while (rc = 0);
    put prod=  qty=;
    rc = iter.next();
  end;
run;
```

次の行が SAS ログに書き出されます。

```
prod=orange qty=329559
prod=apple qty=384223
prod=banana qty=398487
```

関連項目

- “ハッシュ反復子オブジェクトの使用 ”

メソッド:

- “LAST メソッド” (212 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (185 ページ)

LAST メソッド

基となる PROC FCMP ハッシュオブジェクトの最終値を返します。

適用対象: PROC FCMP ハッシュ反復子オブジェクト

構文

rc=*object*.**LAST**( );

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しにリターンコード変数を指定せずにメソッドが失敗した場合、キーが見つからないことを示すエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

LAST メソッドは、ハッシュオブジェクト内の最後のデータアイテムを返します。ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントで **ordered: 'yes'**または **ordered: 'ascending'**引数タグを使用した場合、ハッシュオブジェクト内のデータアイテムはキー値の昇順で並べ替えられるため、'最大'キー(最大の数値または最後のアルファベット文字)のデータアイテムが返されます。反対に、ハッシュオブジェクトをインスタンス化するときに、DECLARE ステートメントで **ordered: 'descending'**引数タグを使用した場合、ハッシュオブジェクト内のデータアイテムはキー値の降順で並べ替えられるため、'最小'キー(最小の数値または最初のアルファベット文字)のデータアイテムが返されます。

ハッシュオブジェクト内の最初のデータアイテムを返すには、FIRST メソッドを使用します。

注: LAST メソッドは、データ変数にデータアイテムの値を設定し、メソッド呼び出し後にそのデータアイテムを使用できるようにします。

関連項目

- “ハッシュ反復子オブジェクトの使用 ”

メソッド:

- “FIRST メソッド” (210 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (185 ページ)

NEXT メソッド

基となる PROC FCMP ハッシュオブジェクトの次の値を返します。

適用対象: PROC FCMP ハッシュ反復子オブジェクト

構文

rc=*object*.**NEXT**( );

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しにリターンコード変数を指定せずにメソッドが失敗した場合、キーが見つからないことを示すエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

NEXT メソッドを繰り返し使用することで、ハッシュオブジェクト内を移動し、データアイテムをキーの順序で返すことができます。

FIRST メソッドは、ハッシュオブジェクト内の最初のデータアイテムを返します。

ハッシュオブジェクト内の前のデータアイテムを返すには、PREV メソッドを使用できます。

注: NEXT メソッドは、データ変数にデータアイテムの値を設定し、メソッド呼び出し後にそのデータアイテムを使用できるようにします。

注: FIRST メソッドを呼び出さずに NEXT メソッドを呼び出した場合でも、NEXT メソッドはハッシュオブジェクト内の最初のアイテムから開始します。

関連項目

- “ハッシュ反復子オブジェクトの使用 ”

メソッド:

- “FIRST メソッド” (210 ページ)
- “PREV メソッド” (215 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (185 ページ)

PREV メソッド

基となる PROC FCMP ハッシュオブジェクトの前の値を返します。

適用対象: PROC FCMP ハッシュ反復子オブジェクト

構文

rc=*object*.**PREV**( );

引数

rc
メソッドが成功したか失敗したかを示します。

0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しにリターンコード変数を指定せずにメソッドが失敗した場合、キーが見つからないことを示すエラーメッセージがログに書き込まれます。ハッシュ反復子は、キーを使ってハッシュテーブルを移動します。

object
ハッシュ反復子オブジェクト名を指定します。

詳細

ハッシュオブジェクトを移動して逆のキー順序でデータアイテムを返すには、PREV メソッドを反復して使用します。

FIRST メソッドは、ハッシュオブジェクト内の最初のデータアイテムを返します。LAST メソッドは、ハッシュオブジェクト内の最後のデータアイテムを返します。

ハッシュオブジェクトの次のデータアイテムを返すには、NEXT メソッドを使用できます。

注: LAST メソッドを呼び出さずに PREV メソッドを呼び出した場合でも、PREV メソッドはハッシュオブジェクト内の最後のアイテムから開始します。

注: PREV メソッドは、データ変数にデータアイテムの値を設定し、メソッド呼び出し後にそのデータアイテムを使用できるようにします。

関連項目

- “ハッシュ反復子オブジェクトの使用 ”

メソッド:

- “FIRST メソッド” (210 ページ)
- “LAST メソッド” (212 ページ)
- “NEXT メソッド” (213 ページ)

ステートメント:

- “DECLARE ステートメント: ハッシュオブジェクトとハッシュ反復子オブジェクト” (185 ページ)

SETCUR メソッド

反復を開始するキーアイテムを 指定します。

適用対象: PROC FCMP ハッシュ反復子オブジェクト

注: PROC FCMP SETCUR メソッドは 2024.07 から使用可能です。

構文

rc=*object*.**SETCUR** (KEY: '*keyvalue-1*'<, ...KEY: '*keyvalue-n*'>);

引数

rc
メソッドが成功したか失敗したかを示します。

ゼロのリターンコードは成功を表し、ゼロ以外の値は失敗を表します。メソッド呼び出しでリターンコード変数を指定せず、そのメソッドが失敗した場合、該当するエラーメッセージがログに書き込まれます。

object
ハッシュ反復子オブジェクト名を指定します。

KEY: '*keyvalue*'
反復の開始キーとしてのキー値を指定します。

詳細

ハッシュ反復子を使うと、ハッシュオブジェクト内のどのアイテムでも反復を開始できます。SETCUR メソッドは、反復の開始キーを設定します。開始アイテムの指定には、KEY オプションを使用します。

例: 開始キーアイテムの 指定

この例では、売上データを含むデータセットを作成します。329559 個以上売れた商品をリストアップしたいとします。データがハッシュオブジェクト内に読み込まれ、データの取得に SETCUR メソッドと NEXT メソッドが使用されます。

```
data work.sales;
  input prod $1-6 qty $9-14;
  datalines;
banana  398487
apple   384223
kiwi    364858
orange  329559
mango   295874
;
proc fcmp;
file log;
    length prod $10 qty $6;
  /* Declare hash object and read SALES data set as ordered */
  if _N_ = 1 then do;
    declare hash h(dataset: 'work.sales', ordered: 'yes');
    declare hiter iter('h');
    /* Define key and data variables */
    rc=h.defineKey('qty');
    rc=h.defineData('prod', 'qty');
    rc=h.defineDone();
    /* avoid uninitialized variable notes */
    call missing(qty, prod);
  end;
  /* Iterate through the hash object and output data values */
  rc = iter.setcur(key: '329559');
  do while (rc = 0);
    put prod=  qty=;
    rc = iter.next();
  end;
run;
```

次の行が SAS ログに書き出されます。

```
prod=orange qty=329559
prod=kiwi qty=364858
prod=apple qty=384223
prod=banana qty=398487
```

8

ディクショナリオブジェクトの使用

FCMP ディクショナリオブジェクトの使用 **219**
ディクショナリオブジェクトの使用 219
はじめに 219

FCMP ディクショナリオブジェクトの使用

ディクショナリオブジェクトの使用

ディクショナリオブジェクトは、PROC FCMP プログラムにデータを格納する別の方法を提供します。ハッシュオブジェクトは、数値や文字のデータのみを格納することに限定されていますが、ディクショナリは、配列、ハッシュオブジェクト、さらには他のディクショナリに加えて、数値や文字のデータを格納できます。ディクショナリは、データを参照または値別に格納できます。ディクショナリにデータを格納するため、データがあらかじめ定義されている必要はありません。

はじめに

個々のメソッドの構文については、“ディクショナリオブジェクトの言語要素”を参照してください。

ディクショナリの作成

DECLARE DICTIONARY ステートメントを使用して、ディクショナリを作成します。また、DNARY という略語は、ディクショナリを作成するための有効なキーワードです。

```
proc fcmp;
  declare dictionary d1;
  declare dnary d2;
run;
```

データの格納

デフォルトでは、数値データと文字データは、値によってディクショナリに 格納されます。配列、ハッシュオブジェクト、ハッシュ反復子、ASTORE オブジェクト、Python オブジェクト、その他のディクショナリは、デフォルトでは参照別にディクショナリに格納されます。データは、これらのいずれかの方法でディクショナリに格納されます。

表 8.1　データ格納メソッド

タスクとメソッド	説明
直接割り当て	データアイテムにディクショナリキーを 割り当てることで、データアイテムを 直接ディクショナリに格納できます。直接割り当てを使用した場合、データは、データの種類に応じて値または参照別に自動的に格納されます。場合によっては、他の格納メソッドを使ってデフォルト動作をオーバーライドできます。 d1[*key*] = *your-data*;
データを値別に格納する (“CLONE メソッド”)	CLONE メソッドは、配列を強制的にディクショナリに値別に格納します。CLONE メソッドは、文字データや数値データをディクショナリに格納する際にも使用できます。ハッシュ、ハッシュ反復子、その他のディクショナリは、CLONE メソッドを使って値を格納することはできません。 rc = d1.clone(*your-data*, *key*);
データを参照別に格納する (“REF メソッド”)	REF メソッドは、数値データと文字データを強制的にディクショナリに参照別に格納します。デフォルトでは、数値データと文字データは値別に格納されます。配列、ハッシュオブジェクト、ハッシュ反復子、その他のディクショナリも、REF メソッドを使ってディクショナリに格納できます。 rc = d1.ref(*your-data*, *key*);

例 1: データをディクショナリに格納する

この例では、データアイテムをディクショナリに 格納するためのすべての方法を示しています。

```
proc fcmp;
  declare dictionary d1;
  array arr[3] 5 10 15; x = 25; y = 'string';
  d1[key-1] = x;
  rc = d1.clone(arr, key-2);
  rc = d1.ref(x, key-3);
run;
```

データの取得

ディクショナリにデータを登録した後、そのデータをディクショナリ外の変数に戻す方法は複数あります。

- データアイテムに割り当てられているキーがわかっている場合は、ディクショナリデータアイテムに変数を直接割り当てます。
- キーを指定せずにディクショナリのデータアイテムを 返したい場合は、表 7.2 データ取得メソッドで説明する他の返し方を使用してください。これらの返し方を使用すると、ディクショナリ内の位置に基づいてデータアイテムをプログラムで返すことができます。ディクショナリデータアイテムを 返すために指定した変数は、コピーしているデータ型と一致していなければならず、そうでなければ変数は missing に設定されます。

ディクショナリに格納されているデータアイテムは、これらいずれかのメソッドを使用して返されます。

表 8.2 データ取得メソッド

タスクとメソッド	説明
直接割り当て	ディクショナリのデータアイテムは、新しい変数をディクショナリのキーと等しく設定することで、新しい変数に直接割り当てることができます。 a = d1[*key*];
最初のアイテムを返す (“FIRST メソッド”)	FIRST メソッドは、ディクショナリの最初のデータアイテムを指定された変数にコピーし、ディクショナリ反復子を最初の位置に設定します。FIRST メソッドを使うと、ディクショナリ反復子が初期化されます。 rc = d1.first(*data-variable*);

タスクとメソッド	説明
次のアイテムを返す(“NEXT メソッド”)	NEXT メソッドは、ディクショナリ反復子を 1 つ進め、その位置にあるデータアイテムのコピーを指定した変数に返します。ディクショナリ反復子が初期化されていない場合、NEXT メソッドは、ディクショナリの最初のアイテムをコピーし、反復子を最初の位置に設定します。 *data-variable* = d1.next();
前のアイテムを返す(“PREV メソッド”)	PREV メソッドは、ディクショナリ反復子を前の位置に移動させ、その位置にあるデータアイテムのコピーを指定した変数に返します。ディクショナリ反復子が初期化されていない場合、PREV メソッドは、ディクショナリの最後のアイテムをコピーし、反復子を最後の位置に設定します。 *data-variable* = d1.prev();
最後のアイテムを返す(“LAST メソッド”)	LAST メソッドは、ディクショナリの最後のデータアイテムを指定された変数にコピーし、ディクショナリ反復子を最初の位置に設定します。LAST メソッドを使うと、ディクショナリ反復子が初期化されます。 rc = d1.last(*data-variable*);

例 2: ディクショナリからデータアイテムを 取得する

この例では、さまざまなデータ取得メソッドをすべて使用して、ディクショナリからデータアイテムを返す方法を示します。

```
proc fcmp;
  declare dictionary d1;
  array arr[3] 5 10 15; array myvar1[3]; array myvar3[3];
  x = 25; y = 'string';
  length myvar2 $6;
  d1[1] = x;
  rc = d1.clone(arr,2);
  rc = d1.ref(y, 3);
  a = d1[1];
  rc = d1.first(myvar1);
  myvar2 = d1.next();
  myvar3 = d1.prev();
  rc = d1.last(myvar4);
  put a= myvar1= myvar2= myvar3= myvar4=;
run;
```

上記のステートメントは、次の結果を生成します。

```
a=25 myvar1[1]=5 myvar1[2]=10 myvar1[3]=15 myvar2=string myvar3[1]=5 myvar3[2]=10
myvar3[3]=15
myvar4=25
```

ディクショナリのナビゲーション

前述のメソッドでは、ディクショナリのアイテムを 循環させながら、アイテムのコピーを指定した変数に返すことができます。アイテムのコピーを返さずにディクショナリのアイテムを循環させたり、アイテムが存在するかどうかを確認する場合は、これらのメソッドを使用します。

表 8.3　ディクショナリをナビゲートするメソッド

タスクとメソッド	説明
次のデータアイテムのチェック (“HASNEXT メソッド”)	HASNEXT メソッドは、ディクショナリ反復子を進めることなく、ディクショナリの先読みを行います。リターンコードは、次のデータアイテムが存在するかどうかを示します。アイテムが存在する場合は、オプションの引数を使用してデータ型を返すことができます。ディクショナリの最初のデータアイテムから開始する場合、HASNEXT メソッドを使用して、ディクショナリの終わりに到達したことを示すことができます。ディクショナリ反復子が初期化されていない場合、HASNEXT メソッドは、ディクショナリの最初のアイテムをチェックします。 *item-found* = d1.hasnext(*data-type-rc*,*array-rc*);
前のデータアイテムのチェック (“HASPREV メソッド”)	HASPREV メソッドは、ディクショナリ反復子を動かすことなく、ディクショナリの後読みを行います。リターンコードは、前のデータアイテムが存在するかどうかを示します。アイテムが存在する場合は、オプションの引数を使用して属性を返すことができます。ディクショナリの最後のデータアイテムから開始する場合、HASPREV メソッドを使用して、ディクショナリの先頭に到達したことを示すことができます。ディクショナリ反復子が初期化されていない場合、HASPREV メソッドは、ディクショナリの最後のアイテムをチェックします。 *item-found* = d1.hasprev(*data-type-rc*, *array-rc*);
反復子を 1 つ前の位置に移動させる(“SKIPNEXT メソッド”)	SKIPNEXT メソッドは、反復子の位置を 1 つ前に進めます。リターンコードは、新しい位置にアイテムが存在するかどうかを示しますが、属性を返すことはできません。 *item-found* = d1.skipnext();
反復子を 1 つ後ろの位置に移動させる(“SKIPPREV メソッド”)	SKIPPREV メソッドは、反復子の位置を 1 つ後ろへ移動させます。リターンコードは、新しい位置にアイテムが存在するかどうかを示しますが、属性を返すことはできません。 *item-found* = d1.skipnext();

例 3: ディクショナリをナビゲートし、ディクショナリ反復子を循環させる

この例では、ディクショナリ反復子を操作して、ディクショナリのどちらかの端に近づいたときにデータアイテムをチェックする方法を示します。

```
proc fcmp;
  declare dictionary d1;
  array arr[3] 5 10 15; x = 25; y = 'string';
  HasNext = .; HasPrev = .;
  ItemFound = .; DType = .; IsArr = .;
  d1[1] = x;
  rc = d1.clone(arr,2);
  rc = d1.ref(y, 3);
  rc = d1.first(myvar1);
  HasNext = d1.hasnext(DType,IsArr);
  put HasNext= Dtype= IsArr=;
  ItemFound = d1.skipnext();
  put ItemFound=;
  ItemFound = d1.skipnext();
  put ItemFound=;
  ItemFound = d1.skipprev();
  put ItemFound=;
  HasPrev = d1.hasprev(DType,IsArr);
  put HasPrev= Dtype= IsArr=;
run;
```

上記のステートメントは、次の結果を生成します。

```
HasNext=1 DType=2 IsArr=0
ItemFound=1
ItemFound=1
ItemFound=1
HasPrev=1 DType=1 IsArr=1
```

ディクショナリのアイテムの 入れ替えと削除

既存のディクショナリキーにデータアイテムを 割り当てると、その場所のデータアイテムが置き換えられます。そのキーに以前格納されていたデータの型は関係ありません。例えば、あるキーに文字列を割り当てた後、同じキーに数値データを割り当てる場合、コードや型を変換する必要はありません。データアイテムの 置き換えは、これまでに説明したデータ格納メソッド(直接割り当て、CLONE、REF)のいずれかを使って行うことができます。次に例を示します。

ディクショナリのデータアイテムの 置き換え

```
proc fcmp;
  declare dictionary d1;
  length a $6;
```

```
    d1[1] = 'string';
    a = d1[1];
    put a=;
    d1[1] = 25;
    b = d1[1];
    put b=;
  run;
```

上記のステートメントは、次の結果を生成します。

```
a=string
b=25
```

表 8.4　ディクショナリからデータアイテムを 削除する

タスクとメソッド	説明
指定したデータアイテムまたは複数のデータアイテムを削除する(“REMOVE メソッド”)	REMOVE メソッドは、ディクショナリから 1 つまたは複数の指定されたデータアイテムを削除します。指定されたキーと、指定されたキーに格納されているすべてのデータがディクショナリから削除されます。 rc = d1.remove(*key-1*, *key-2*);
すべてのデータアイテムをクリアする(“CLEAR メソッド”)	CLEAR メソッドは、ディクショナリに格納されているすべてのキーとデータを削除します。ディクショナリオブジェクト自体は削除されず、新しいデータを格納するために使用できます。 rc = d1.clear();

ディクショナリのアイテムを 数えて記述する

ディクショナリの属性とデータアイテムの属性は、これらのメソッドを使って返されます。

表 8.5　ディクショナリデータアイテムの 属性の特定

タスクとメソッド	説明
アイテムの数を数える(“NUM_ITEMS メソッド”)	NUM_ITEMS メソッドは、ディクショナリに格納されているアイテムの数を返します。関数やループを通してデータがディクショナリに割り当てられている場合、NUM_ITEMS メソッドは、ディクショナリの予想アイテム数が正しいかどうかをすばやく検証できます。 *n-items* = d1.num_items();

タスクとメソッド	説明
指定されたキーでデータを記述する (“DESCRIBE メソッド”)	DESCRIBE メソッドは、指定されたキーに格納されているデータの型と、そのデータが配列であるかどうかを返します。これらの戻り値は、ディクショナリからデータアイテムを 返すために必要な変数データ型をプログラムで決定するために使用できます。 *data-type-rc* = d1.describe(*array-rc*, *key*);

例 4: ディクショナリ属性を特定する

この例では、異なるディクショナリの属性を返す方法を示します。

```
proc fcmp;
  declare dictionary d1;
  array arr[3] 5 10 15;
  DType = .; IsArr = .;
  x = 25; y = 'string';
  d1[1] = x;
  rc = d1.clone(arr,2);
  rc = d1.ref(y, 3);
  n = d1.num_items();
  DType = d1.describe(IsArr, 2);
  put n= Dtype= IsArr=;
run;
```

上記のステートメントは、次の結果を生成します。

```
n=3 DType=1 IsArr=1
```

9

ディクショナリオブジェクトの言語要素

ディクショナリ **227**
CLEAR メソッド 227
CLONE メソッド 228
DECLARE ステートメント: ディクショナリオブジェクト 230
DESCRIBE メソッド 231
FIRST メソッド 233
HASNEXT メソッド 235
HASPREV メソッド 237
LAST メソッド 239
NEXT メソッド 241
NUM_ITEMS メソッド 242
PREV メソッド 243
REF メソッド 244
REMOVE メソッド 246
SKIPNEXT メソッド 247
SKIPPREV メソッド 248

ディクショナリ

CLEAR メソッド

ディクショナリからキーと値のペアをすべて削除します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用を参照してください。

構文

rc=*object-reference*.**CLEAR**();

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
オブジェクトの名前を指定します。

詳細

CLEAR メソッドは、ディクショナリオブジェクト内部に格納されているすべてのキーとデータを削除します。ディクショナリオブジェクト自体は削除されず、クリア後も新しいキーやデータを格納するために使用できます。

例: ディクショナリオブジェクトのクリア

```
proc fcmp;
  declare dictionary d;
  Num = .;
  d[1] = 25;
  Num = d.num_items();
  put Num=;
  rc = d.clear();
  Num = d.num_items();
  put Num=;
run;
```

上記のステートメントは、次の結果を生成します。

```
Num=1
Num=0
```

CLONE メソッド

配列を値ごとに格納します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用を参照してください。

構文

rc=*object-reference*.**CLONE**(*data*, *key-1* <, ...*key-n*>);

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
オブジェクトの名前を指定します。

data
指定されたキーとペアになるデータを指定します。

key-1
データを配置する場所を示すキー値を指定します。キーは、数値や文字のリテラルや変数で表現できます。少なくとも 1 つのキーを提供する必要があります。

オプション引数

key-n
データが配置される場所を示す追加のキー値を指定します。1 つのディクショナリには最大 6 つのキーを格納することができます。

詳細

パフォーマンス上の理由から、配列、ハッシュオブジェクト、ディクショナリなどの複雑なデータタイプは、デフォルトでは参照別にディクショナリ内に格納されます。CLONE メソッドは、配列を値ごとに格納します。ディクショナリは、値によるハッシュオブジェクト、ハッシュ反復子、およびディクショナリの保存をサポートしていません。

例

```
proc fcmp;
  declare dnary d;
  array a[10]; array b[10];
  do i= 1 to 10;
   a[i] = i**2;
  end;
  rc = d.clone(a, 5);
  a[1] = 17;
  b = d[5];
  put b=;
run;
```

上記のステートメントは、次の結果を生成します。

```
b[1]=1 b[2]=4 b[3]=9 b[4]=16 b[5]=25 b[6]=36 b[7]=49 b[8]=64 b[9]=81 b[10]=100
```

DECLARE ステートメント: ディクショナリオブジェクト

ディクショナリオブジェクトを宣言します。

別名: DCL

適用対象: ディクショナリオブジェクト

参照項目: 詳細については ディクショナリオブジェクトの使用を参照してください。

構文

DECLARE DICTIONARY *object-reference*;

必須引数

object-reference
ディクショナリオブジェクトの参照名を指定します。

詳細

PROC FCMP プログラムでディクショナリオブジェクトを作成するには、キーワード DICTIONARY のある DECLARE ステートメントを使用します。ディクショナリオブジェクトを作成するためのキーワードとしては、DNARY という略語も有効です。

例

```
proc fcmp;
  declare dictionary d1;
  declare dnary d2;
  n1 = d1.num_items();
  n2 = d2.num_items();
  put n1=; put n2=;
run;
```

上記のステートメントは、次の結果を生成します。

```
n1=0
n2=0
```

DESCRIBE メソッド

ある場所のディクショナリに格納されているデータの情報を返します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については ディクショナリオブジェクトの使用を参照してください。

構文

data-type=*object-reference*.**DESCRIBE**(*array-indicator*, *key* <, ...,*key-n*>);

必須引数

data-type
見つかったデータ型を数値で格納する変数を指定し、データがない場合は MISSING を指定します。

object-reference
オブジェクトの名前を指定します。

array-indicator
配列の指標となる値を格納する変数を指定します。データがない場合は、値は MISSING に設定されます。

key
どのデータアイテムを記述するかを示すキー値を指定します。

詳細

DESCRIBE メソッドは、ディクショナリの指定された場所に格納されているデータに関する情報を返します。リターンコードの引数 *data-type* は、データタイプを示す数値に設定されます。*array-indicator* 引数は、データが配列であるかどうかを示す数値に設定されます。次のテーブルは、データ型と配列インジケーターのリターンコード値に関する情報を示します。

表 9.1 データ型リターンコード

リターンコード	データ型
1	double
2	char
0	MISSING

リターンコード	データ型
-1	ディクショナリ
-2	ハッシュオブジェクト
-3	ハッシュ反復子
-4	その他のオブジェクト
-5	ASTORE オブジェクト
-6	Python オブジェクト

表 9.2　配列インジケーターのリターンコード

リターンコード	説明
1	指定された場所のデータは配列になります。
0	指定された場所のデータは配列ではありません。
MISSING	指定された場所にデータがありません。

例: ディクショナリから属性を返すための DESCRIBE メソッドの使用

```
proc fcmp;
  declare dictionary d;
  delcare dictionary d2;
  DType = .; IsArr = .;
  array Arr{3} 5 10 15;
  d[1] = 25; d[2] = 'char-string'; d[3] = Arr; d[4] = d2;
  do i = 1 to 4;
   DType = d.describe(IsArr, i);
   put DType=;
   put IsArr=;
  end;
run;
```

上記のステートメントは、次の結果を生成します。

```
dtype=1
isarr=0
dtype=2
isarr=0
dtype=1
isarr=1
dtype=-1
isarr=0
```

FIRST メソッド

ディクショナリの最初のアイテムを指定した変数にコピーし、反復子を最初の位置に設定します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については ディクショナリオブジェクトの使用を参照してください。

構文

rc=*object-reference*.**FIRST**(*data-variable*<, *data-type*, *array-indicator*>);

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
オブジェクトの名前を指定します。

data-variable
ディクショナリの最初のアイテムのコピーを格納する変数を指定します。指定された *data-variable* が、ディクショナリの最初のアイテムと同じデータ型でない場合、*data-variable* 変数は MISSING に設定されます。

オプション引数

data-type
見つかったデータ型を数値で格納する変数を指定し、データがない場合は MISSING を指定します。

array-indicator
配列の指標となる値を格納する変数を指定します。データがない場合は、値は MISSING に設定されます。

詳細

FIRST メソッドは、ディクショナリの最初のデータアイテムのコピーを指定された変数に格納し、ディクショナリ反復子をディクショナリの最初の位置に設定します。コピーは第一引数の *data-variable* に格納されます。引数 *data-variable* は、ディクショナリデータアイテムと同じデータ型でなければならず、そうでなければ missing に設定されます。オプションの引数 *data-type* と *array-indicator* を使うことで、最初のデータアイテムに関する追加情報を返すことができます。

注: FIRST メソッドが返すデータアイテムは、プログラム内のディクショナリに順次格納されている最初のアイテムとは一致しません。ディクショナリ内のデータアイテムの順序は連続したものではなく、ディクショナリ内のデータアイテムの変更に伴って変化する可能性があります。

オプション引数は、これらのリターンコードを使用します。

表 9.3　データ型リターンコード

リターンコード	データ型
1	double
2	char
0	MISSING
-1	ディクショナリ
-2	ハッシュオブジェクト
-3	ハッシュ反復子
-4	その他のオブジェクト
-5	ASTORE オブジェクト
-6	Python オブジェクト

表 9.4　配列インジケーターのリターンコード

リターンコード	説明
1	指定された場所のデータは配列になります。
0	指定された場所のデータは配列ではありません。
MISSING	指定された場所にデータがありません。

例: FIRST メソッドでディクショナリの最初のアイテムを返す

```
proc fcmp;
  declare dictionary d;
  DFirst = .; DType = .;IsArr = .;
  d[1] = 25; d[2] = 'test-string'; d["key"] = 534;
  rc = d.first(DFirst, DType, IsArr);
  put DFirst= DType= IsArr=;
run;
```

上記のステートメントは、次の結果を生成します。

```
DFirst=534 DType=1 IsArr=0
```

HASNEXT メソッド

ディクショナリに次のデータアイテムがあるかどうかを示します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については ディクショナリオブジェクトの使用を参照してください。

構文

item-found=*object-reference*.**HASNEXT**(<*data-type*, *array-indicator*>);

必須引数

item-found
次のデータアイテムがディクショナリに 存在するかどうかを返す代入変数を指定します。この変数は、次のデータアイテムがディクショナリに 存在する場合は 1 に、存在しない場合は 0 に設定されます。

object-reference
オブジェクトの名前を指定します。

オプション引数

data-type
見つかったデータ型を数値で格納する変数を指定し、データがない場合は MISSING を指定します。

array-indicator
配列の指標となる値を格納する変数を指定します。データがない場合は、値は MISSING に設定されます。

詳細

注: HASNEXT メソッドは、ディクショナリ反復子の位置を変更しません。

HASNEXT メソッドは、指定されたディクショナリが、現在の反復子の値を超えて次のデータアイテムを持っているかどうかを示します。反復子が初期化されていない場合、HASNEXT メソッドは、ディクショナリの最初のデータアイテムをチェックします。オプションの引数 *data-type* と *array-indicator* を使って、データアイテムに関する追加情報を返すことができます。次のテーブルは、データ型と配列インジケーターのリターンコード値に関する情報を示します。

表 9.5　データ型リターンコード

リターンコード	データ型
1	double
2	char
0	MISSING
-1	ディクショナリ
-2	ハッシュオブジェクト
-3	ハッシュ反復子
-4	その他のオブジェクト
-5	ASTORE オブジェクト
-6	Python オブジェクト

表 9.6　配列インジケーターのリターンコード

リターンコード	説明
1	指定された場所のデータは配列になります。
0	指定された場所のデータは配列ではありません。
MISSING	指定された場所にデータがありません。

例: ディクショナリでの HASNEXT メソッドの使用

```
proc fcmp;
  declare dictionary d;
  FirstItem = .; ItemFound = .; DType = .; IsArr = .;
  d[1] = 25; d[2] = 'string'; d["key"] = 534;
  rc = d.first(FirstItem);
  ItemFound = d.hasnext(DType, IsArr);
  put ItemFound= DType= IsArr=;
run;
```

上記のステートメントは、次の結果を生成します。

```
ItemFound=1 DType=2 IsArr=0
```

HASPREV メソッド

ディクショナリに前のデータアイテムがあるかどうかを示します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用 (219 ページ)を参照してください。

構文

item-found=*object-reference*.**HASPREV**(<*data-type*, *array-indicator*>);

必須引数

item-found
前のデータアイテムがディクショナリに 存在するかどうかを返す代入変数を指定します。この変数は、前のデータアイテムがディクショナリに 存在する場合は 1 に、存在しない場合は 0 に設定されます。

object-reference
オブジェクトの名前を指定します。

オプション引数

data-type
見つかったデータ型を数値で格納する変数を指定し、データがない場合は MISSING を指定します。

array-indicator
配列の指標となる値を格納する変数を指定します。データがない場合は、値は MISSING に設定されます。

詳細

注: HASPREV メソッドは、ディクショナリ反復子の位置を変更しません。

HASPREV メソッドは、指定されたディクショナリに、現在の反復子の前に前のデータアイテムがあるかどうかを示します。反復子が初期化されていない場合、HASPREV メソッドは、ディクショナリの最後のデータアイテムをチェックします。オプションの引数 *data-type* と *array-indicator* を使って、データアイテムに関する追加情報を返すことができます。次のテーブルは、データ型と配列インジケーターのリターンコード値に関する情報を示します。

表 9.7　データ型リターンコード

リターンコード	データ型
1	double
2	char
0	MISSING
-1	ディクショナリ
-2	ハッシュオブジェクト
-3	ハッシュ反復子
-4	その他のオブジェクト
-5	ASTORE オブジェクト
-6	Python オブジェクト

表 9.8　配列インジケーターのリターンコード

リターンコード	説明
1	指定された場所のデータは配列になります。
0	指定された場所のデータは配列ではありません。
MISSING	指定された場所にデータがありません。

例: HASPREV メソッドを使ってディクショナリの前のアイテムを調べる

```
proc fcmp;
  declare dictionary d;
  LastItem = .; ItemFound = .; DType = .; IsArr = .;
  d[1] = 25; d[2] = 'test-string'; d["key"] = 534;
  rc = d.last(LastItem);
  ItemFound = d.hasprev(DType, IsArr);
  put ItemFound= DType= IsArr=;
run;
```

上記のステートメントは、次の結果を生成します。

```
ItemFound=1 DType=2 IsArr=0
```

LAST メソッド

ディクショナリの最後のアイテムを指定した変数にコピーし、反復子を最初の位置に設定します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用 (219 ページ)を参照してください。

構文

rc=*object-reference*.**LAST**(*data-variable* <, *data-type*, *array-indicator*>);

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
オブジェクトの名前を指定します。

data-variable
ディクショナリの最後のアイテムのコピーを格納する変数を指定します。指定された *data-variable* が最初のアイテムと同じデータ型でない場合、*data-variable* 変数は missing に設定されます。

オプション引数

data-type
見つかったデータ型を数値で格納する変数を指定し、データがない場合は MISSING を指定します。

array-indicator
配列の指標となる値を格納する変数を指定します。データがない場合は、値は MISSING に設定されます。

詳細

LAST メソッドは、ディクショナリの最後のデータアイテムのコピーを指定された変数に格納し、ディクショナリ反復子をディクショナリの最後の位置に設定します。コピーは、引数の *data-variable* に格納されます。指定された変数が、ディクショナリの最後のデータアイテムと同じデータ型でない場合、その変数は missing に設定されます。オプションの引数 *data-type* と *array indicator* を使うことで、最後のデータアイテムに関する追加情報を返すことができます。

注: LAST メソッドが返すデータアイテムは、プログラム内のディクショナリに順次格納されている最後のアイテムとは一致しません。ディクショナリ内のデータアイテムの順序は連続したものではなく、ディクショナリ内のデータアイテムの変更に伴って変化する可能性があります。

次のテーブルは、データ型と配列インジケーターのリターンコード値に関する情報を示します。

表 9.9　データ型リターンコード

リターンコード	データ型
1	double
2	char
0	MISSING
-1	ディクショナリ
-2	ハッシュオブジェクト
-3	ハッシュ反復子
-4	その他のオブジェクト
-5	ASTORE オブジェクト
-6	Python オブジェクト

表 9.10 配列インジケーターのリターンコード

リターンコード	説明
1	指定された場所のデータは配列になります。
0	指定された場所のデータは配列ではありません。
MISSING	指定された場所にデータがありません。

例: LAST メソッドを使ってディクショナリの最後のアイテムのコピーを返す

```
proc fcmp;
  declare dictionary d;
  DLast = .; DType = .; IsArr = .;
  d[1] = 25; d[2] = 'test-string'; d["key"] = 534;
  rc = d.last(DLast, DType, IsArr);
  put DLast= DType= IsArr=;
run;
```

上記のステートメントは、次の結果を生成します。

```
DLast=25 DType=1 IsArr=0
```

NEXT メソッド

反復子を 1 つ前進させ、その位置にあるデータアイテムをコピーします。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用を参照してください。

構文

data-variable=*object-reference*.**NEXT**();

必須引数

data-variable
ディクショナリの次のアイテムのコピーを格納する変数を指定します。変数は、ディクショナリデータアイテムと同じデータ型でなければならず、そうでなければ変数は missing に設定されます。

object-reference
オブジェクトの名前を指定します。

詳細

NEXT メソッドは、ディクショナリの次のデータアイテムのコピーを代入変数 *data-variable* に返し、反復子をディクショナリの次の位置に進めます。指定された *data-variable* は、ディクショナリデータアイテムと 同じデータ型でなければならず、そうでなければ *data-variable* 変数は missing に設定されます。反復子が初期化されていない場合は、ディクショナリの最初のデータアイテムが返されます。ディクショナリに次のデータアイテムが存在しない場合は、リターン変数は missing に設定されます。

例

```
proc fcmp;
  declare dictionary d;
  length NextVar $8;
  array Arr[5];
  array FirstVar[5];
  do i =1 to 5;
   Arr[i] = i**2;
  end;
  d[1] = 25; d[2] = 'string'; d["key"] = arr;
  rc = d.first(FirstVar);
  put FirstVar=;
  NextVar = d.next();
  put NextVar=;
run;
```

上記のステートメントは、次の結果を生成します。

```
FirstVar[1]=1 FirstVar[2]=4 FirstVar[3]=9 FirstVar[4]=16 FirstVar[5]=25
NextVar=string
```

NUM_ITEMS メソッド

ディクショナリ内のデータアイテム数を返します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用を参照してください。

構文

number-items=*object-reference*.**NUM_ITEMS**();

必須引数

number-items
メソッドが返すアイテムの数を格納する変数を指定します。

object-reference
オブジェクトの名前を指定します。

詳細

NUM_ITEMS メソッドは、ディクショナリに格納されているデータアイテムの数を返します。アイテムの数は、代入変数 *number-items* で返されます。

例

```
proc fcmp;
  declare dictionary d;
  n = .;
  d[1] = 25; d[2] = 'string';
  n = d.num_items();
  put n=;
run;
```

上記のステートメントは、次の結果を生成します。

```
n=2
```

PREV メソッド

ディクショナリの前のデータアイテムのコピーを返し、反復子を前の位置に移動させます。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用を参照してください。

構文

data-variable=*object-reference*.**PREV**();

必須引数

data-variable
ディクショナリの前のデータアイテムのコピーを格納する変数を指定します。変数は、ディクショナリデータアイテムと同じデータ型でなければならず、そうでなければ変数は missing に設定されます。

object-reference
オブジェクトの名前を指定します。

詳細

PREV メソッドは、ディクショナリの前のデータアイテムのコピーを代入変数 *data-variable* に格納し、可能であれば反復子を 1 つ後ろの位置に移動させます。反復子が初期化されていない場合、PREV メソッドは、ディクショナリの最後のデータアイテムを返します。

例

```
proc fcmp;
  declare dictionary d;
  array Arr[5];
  do i =1 to 5;
   Arr[i] = i**2;
  end;
  PrevVar = '      '; LastVar = .;
  d[1] = 25; d[2] = 'string'; d["key"] = Arr;
  rc = d.last();
  put LastVar=;
  PrevVar = d.prev();
  put PrevVar=;
run;
```

上記のステートメントは、次の結果を生成します。

```
LastVar=25
PrevVar=string
```

REF メソッド

数値や文字の変数を参照別に格納します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用を参照してください。

構文

rc=*object-reference*.**REF**(*data*, *key-1* ...< *key-n*>);

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
オブジェクトの名前を指定します。

data
指定されたキーとペアになるデータを指定します。

key-1
データを配置する場所を示すキー値を指定します。キーは、数値や文字のリテラルや変数で表現できます。少なくとも 1 つのキーを提供する必要があります。

オプション引数

key-n
データが配置される場所を示す追加のキー値を指定します。1 つのディクショナリには最大 6 つのキーを格納することができます。

詳細

REF メソッドは、数値データと文字データを参照別に格納します。デフォルトでは、ディクショナリは数値データと文字データを値別に格納します。

例: REF メソッドを使って参照別にデータを格納する

```
proc fcmp;
  declare dictionary d;
  RefVar =.; NumVar = 25;
  rc = d.ref(NumVar, 1);
  RefVar = d[1];
  put RefVar=;
  NumVar = 30; RefVar = d[1];
  put RefVar=;
run;
```

上記のステートメントは、次の結果を生成します。

```
RefVar= 25
RefVar= 30
```

REMOVE メソッド

ディクショナリから指定したキーと値のペアを削除します。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用を参照してください。

構文

rc=*object-reference*.**REMOVE**(*key-1* <, ...,*key-n*>);

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
オブジェクトの名前を指定します。

key-1
ディクショナリから削除するキーを指定します。

オプション引数

key-n
ディクショナリから削除する追加のキーを指定します。

詳細

REMOVE メソッドは、ディクショナリから指定したキーとデータのペアを削除します。

例: REMOVE メソッドでディクショナリからアイテムを削除する

```
proc fcmp;
  declare dictionary d;
  array Arr[5];
  do x = 1 to 5;
   Arr[x] = x**2;
  end;
```

```
    n = .;
    d[1] = 25; d[2] = 'string'; d[3] = Arr;
    n = d.num_items();
    put n=;
    rc = d.remove(1);
    put rc=;
    n = d.num_items();
    put n=;
  run;
```

上記のステートメントは、次の結果を生成します。

```
N=3
rc=0
N=2
```

SKIPNEXT メソッド

可能であれば、反復子の位置を 1 つ前に進めます。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用を参照してください。

構文

item-found=*object-reference*.**SKIPNEXT**();

必須引数

item-found
次のデータアイテムがディクショナリに 存在するかどうかを示す値を返します。次のデータアイテムが存在する場合は 1 を、ディクショナリの終わりに達している場合は 0 を返します。

object-reference
オブジェクトの名前を指定します。

詳細

SKIPNEXT メソッドは、反復子の位置を 1 つ前に進めます。代入変数 *item-found* は、反復子が 1 つ進んだかどうか、次のアイテムがディクショナリに 存在するかどうかを示します。

例: SKIPNEXT メソッドを使ってディクショナリ反復子を進め、ディクショナリ内のアイテムをチェックする

```
proc fcmp;
  declare dictionary d;
  DFirst = .; ItemFound = .; array Arr[5];
  do x = 1 to 5;
   arr[x] = x**2;
  end;
  d[4] = 25; d[5] = 'string'; d[6] = Arr;
  rc = d.first(DFirst);
  ItemFound = d.skipnext();
  put ItemFound=;
run;
```

上記のステートメントは、次の結果を生成します。

```
ItemFound = 1
```

SKIPPREV メソッド

可能であれば、反復子を前の位置に移動させます。

適用対象: ディクショナリオブジェクト

参照項目: 詳細については、ディクショナリオブジェクトの使用を参照してください。

構文

item-found=*object-reference*.**SKIPPREV**();

必須引数

item-found
次のデータアイテムがディクショナリに 存在するかどうかを示す値を返します。次のデータアイテムが存在する場合は 1 を、ディクショナリの終わりに達している場合は 0 を返します。

object-reference
オブジェクトの名前を指定します。

詳細

SKIPPREV メソッドは、可能であれば、反復子を 1 つ前の位置に後退させます。代入変数 *item-found* は、反復子が 1 つ後退したかどうか、その位置でデータアイテムがディクショナリに存在するかどうかを示します。

例: SKIPPREV メソッドを使って反復子を後退させ、ディクショナリ内のアイテムをチェックする

```
proc fcmp;
  declare dictionary d;
  Dlast = .; ItemFound = .; array Arr[5];
  do x = 1 to 5;
    Arr[x] = x**2;
  end;
  d[4] = 25; d[5] = 'string'; d[6] = Arr;
  rc = d.last(Dlast);
  ItemFound = d.skipprev();
  put ItemFound=;
run;
```

上記のステートメントは、次の結果を生成します。

```
ItemFound = 1
```

10

Python オブジェクトの使用

PROC FCMP Python オブジェクトの使用 **251**
マルチスレッド環境でステートフルデータを使用する Python オブジェクトの操作 251
PROC FCMP Python オブジェクトを使う理由 252
要件 252
制限事項 252
PROC FCMP における Python 関数のワークフロー 252
サブミットメソッドの選択 254
Python オブジェクトからソースコードをクリアする 254
DATA ステップでの Python 関数の使用 255
Python オブジェクトと PROC FCMP Python オブジェクトの違い 256

PROC FCMP Python オブジェクトの使用

マルチスレッド環境でステートフルデータを使用する Python オブジェクトの操作

ステートフルデータを使用する Python コードは、マルチスレッド SAS 環境で実行すると、予期しない結果を返すことがあります。マルチスレッド SAS 環境で Python オブジェクトを使用すると、複数の Python インタープリターセッションがスレッドごとに 1 つずつ起動され、Python コードが実行されます。複数の Python インタープリターセッションが使用されるシナリオで返される結果は、グローバルデータまたはその他のステートフルコンストラクトが使用されている場合、予測できない可能性があります。マルチスレッド SAS 環境で Python コードを実行する場合は、ステートフルな Python プログラムを避けるように注意する必要があります。例については、“ステートフル Python プログラム”を参照してください。

PROC FCMP Python オブジェクトを使う理由

PROC FCMPPython オブジェクトは、Python 関数を SAS プログラムに埋め込んだり、インポートしたりすることができます。Python コードは SAS コードに変換されません。その代わり、Python コードは選択した Python インタープリターで実行され、SAS に結果を返します。ちょっとした Python のコード変更で、SAS から Python 関数を実行し、同時に簡単に両方の言語でプログラミングできるようになりました。

要件

Python オブジェクトは、SAS 環境で Python オブジェクトを使用する前に、環境変数を設定する必要があります。環境変数が設定されていなかったり、間違って設定されていたりすると、Python コードをパブリッシュする際に SAS はエラーを返します。環境変数関連のエラーは、次の例のようになります。

ERROR: MAS_PYPATH environment variable is undefined.

ERROR: The executable *C:\file-path\python.exe* cannot be located
or is not a valid executable.

Python コードのパブリッシュ時にエラーが発生した場合は、サイト管理者にお問い合わせください。サイト管理者は、Deployment Variables for Python に記載されている設定手順に従うことで、SAS 環境で Python オブジェクトを使用できるようになります。

制限事項

FCMP Python オブジェクトは、Python デコレーターをサポートしていません。

PROC FCMP における Python 関数のワークフロー

個々のメソッドの構文の説明については、Python オブジェクトの言語要素を参照してください。

ここでは、Python オブジェクトを PROC FCMP で使用する際の典型的なワークフローを紹介します。

1 Python オブジェクトとディクショナリオブジェクトを宣言します。

- DECLARE ステートメントを使用して、python 型のオブジェクトを宣言します。Python オブジェクトは、Python のソースコードを格納するために使用されます。次に、例を示します。

```
declare object py(python);
```

2 SAS に Python のソースコードを挿入します。

- Python のソースコードを Python オブジェクトに直接読み込むためのサブミットメソッドを選択します。SUBMIT INTO ステートメント使用の例を次に示します。

```
proc fcmp;
declare object py(python);
submit into py;
def MyPyFunc(var1, var2):
   "Output: MyOutputKey"
   MyPyResult = var1 * var2
   return MyPyResult
endsubmit;
run;
```

3 Python のソースコードをパブリッシュします。

- PUBLISH メソッドを使用して、Python オブジェクトに格納されている Python コードを Python インタープリターにサブミットしてコンパイルします。次に、例を示します。

```
rc = py.publish();
```

4 Python のソースコードを呼び出します。

- コールメソッドを使って、パブリッシュされている Python 関数を実行し、その結果をディクショナリに格納します。次に、例を示します。

```
rc = py.call("MyPyFunc", var1, var2);
```

5 ディクショナリの結果を返します。

- Python の関数が返した結果は、ディクショナリに格納されます。デフォルトでは、出力は Python のオブジェクトである RESULTS メンバーディクショナリに格納されます。ディクショナリから結果を返すには、Python 関数の出力文字列から出力キーをディクショナリキーとして指定します。次に、例を示します。

```
MyResult = py.results["MyOutputKey"];
```

これまでの手順で、Python のオブジェクトを使った PROC FCMP プログラムの完成形は、次の例のようになります。

```
proc fcmp;
declare object py(python);
submit into py;
def PyProduct(var1, var2):
   "Output: MyKey"
   newvar = var1 * var2
   return newvar,
endsubmit;
rc = py.publish();
rc = py.call("PyProduct", 5, 10);
MyResult = py.results["MyKey"];
put MyResult=;
run;
```

上記のステートメントは、次の結果を生成します。

```
MyResult=50
```

サブミットメソッドの選択

> 重要　Python オブジェクトサブミットメソッドを使用してサブミットされるすべての Python コードは、Python インタープリターで必要なインデントと空白文字の規則に従う必要があります。

Python のソースコードは、複数の方法で PROC FCMP にサブミットできます。1 つのプログラムで複数のコードサブミットを使用することは可能です。コードを追加でサブミットするたびに、前回のサブミットの後に追加されます。どちらの方法を使うかは、次のような違いがあります。

- SUBMIT INTO ステートメント を使用すると、埋め込まれた Python コードブロックをサブミットしたり、Python ソースコードファイルのファイルパスをサブミットすることができます。ファイルパスオプションの使い方は、INFILE メソッドの使い方とは異なります。Python コードは SAS の関数データセットに格納され、FCMP の解析時にサブミットされます。
- APPEND メソッド は、埋め込まれた SUBMIT INTO メソッドと似ていますが、APPEND メソッドは 1 行のコードのみをサブミットします。Python コードは SAS の関数データセットに格納され、実行時にサブミットされます。コードは実行時にサブミットされるため、変数をコードの行として使用することが可能です。
- INFILE メソッド は、Python のソースコードファイルのファイルパスを指定することができます。ファイルパスの参照のみが SAS 関数のデータセットに格納され、Python ソースコードは格納されません。参照は文字列リテラルとして保存されます。FCMP の解析時にコードをサブミットしています。
- RTINFILE メソッドは、Python ソースコードではなく、SAS 関数データセットのファイルパス参照のみを格納する点で、INFILE メソッドと似ています。しかし、このコードは FCMP の解析時ではなく、ランタイムでサブミットされます。コードは実行時にサブミットされるため、ファイルパスの参照に変数を使用することが可能です。参照は文字列リテラルとして保存されます。

Python オブジェクトからソースコードをクリアする

CALL CLEAR メソッドを使用して、Python オブジェクトに格納されたソースコードとファイル参照を削除し、Python の結果ディクショナリをクリアします。Python

オブジェクトは削除されず、新しいソースコードをオブジェクトにサブミットすることができます。

DATA ステップでの Python 関数の使用

Python オブジェクトと Python オブジェクトのすべてのメソッドは、PROC FCMP ステートメントの中でのみ有効です。例えば、DATA ステッププログラムで Python オブジェクトを宣言しようとすると、エラーになります。しかし、Python 関数を含む PROC FCMP 関数やサブルーチンを作成することで、DATA ステップから Python 関数を呼び出すことができます。Python 関数を含む PROC FCMP 関数は、DATA ステップで有効であり、PROC FCMP で作成された他の関数と同様に呼び出すことができます。次に例を示します。

```
proc fcmp outlib=work.fcmp.pyfuncs;
function MyPyFunc(FCMParg);
declare object py(python);
submit into py;
def TimesFive(PythonArg):
   "Output: MyKey"
   newvar = PythonArg * 5
   return newvar
endsubmit;
rc = py.publish();
rc = py.call("TimesFive",FCMParg);
MyFCMPResult = py.results["MyKey"];
return(MyFCMPResult);
endsub;
run;

options cmplib=work.fcmp;
data _null_;
  x = MyPyFunc(5);
  put x=;
run;
```

上記のステートメントは、次の結果を生成します。

```
x= 25
```

Python オブジェクトと PROC FCMP Python オブジェクトの違い

タプルと出力キー

PROC FCMP で呼び出された Python 関数は、Python タプルで値を返します。そして、Python タプルのアイテムは、Python オブジェクトメンバーディクショナリの個別の要素として、RESULTS ディクショナリ または指定された PROC FCMP ディクショナリを使用して格納されます。Python オブジェクトの CALL メソッドを使って呼び出され、値を返す Python 関数では、Python 関数本体の最初の行が出力文字列の行でなければなりません。出力文字列は、二重引用符で始まり、二重引用符で終わり、"Output: "のみを含む必要があり、続いてコンマで区切られた出力キーのリストが表示されます。キーは、関数が返しているタプルの要素に対応しています。出力キーは、Python タプルからの出力にアクセスするために使用されるディクショナリキーになります。return ステートメントのタプル要素のアイテムごとに、出力キーが必要です。出力キーの順番は、Python タプルの要素の順番と一致します。この例では、出力文字列の詳細を示しています。

Tuple_Element1 の値はディクショナリの *Python_Return_Key1* に格納され、*Tuple_Element2* は *Python_Return_Key2* に格納されています。

```
def MyFunction(foo):
   "Output: Python_Return_Key1, Python_Return_Key2"
   Tuple_Element1 = foo * 2
   Tuple_Element2 = foo + 2
   return Tuple_Element1, Tuple_Element2
```

関数からの出力にアクセスするには、Python のオブジェクトメンバーディクショナリを使用するか、ディクショナリを指定します。

メンバー変数のディクショナリを使用した出力へのアクセス:

```
My_Output1 = py.results["Python_Return_Key1"];
My_Output2 = py.results["Python_Return_Key2"];
```

指定したディクショナリを使用した出力へのアクセス:

```
My_Output1 = My_Dictionary["Python_Return_Key1"];
My_Output2 = My_Dictionary["Python_Return_Key2"];
```

行サイズ制限

PROC FCMP でサブミットする Python コード行は、255 バイト以下でなければなりません。関数内に 255 バイトを超える行がある場合は、Python の行継続文字である"\"(バックスラッシュ)を使って行を継続できます。次に、例を示します。

```
def MyPythonFunc (arg1, arg2, arg3):
```

```
"Output: MyOutputKey"
Result = arg1 + arg2 - arg3 + \
arg2 * arg1
return Result,
```

タイプ変換

SAS は、値を返す際に、Python コードで作成されたデータ値を自動的に SAS のデータ型に変換します。次の変換表は、SAS でどのような Python データ型がサポートされているか、またどのような Python データ型が変換されるかを示しています。変換表に記載されていない Python データ型は、PROC FCMP ではサポートされていません。

表 10.1 Python から SAS への変換表

Python のデータ型	変換された SAS のデータ型
NoneType*	Double*
String	Fixed Character
List of Strings**	Fixed Character Array**
Integer	Double
List of Integers**	Double Array**
Long	Double
List of Longs**	Double Array**
Double	Double
List of Doubles**	Double Array**
日付	Double
Date Array**	Double Array**
時間	Double
List of Times**	Double Array**
DateTime	Double
List of DateTimes**	Double Array**
その他の演算子	非対応

* 任意の型の Python 変数は、NoneType 型に変更できます。NoneType は None という値を持ち、値を持たない変数のような特別な条件を示します。この特別な条件は、SAS の"missing"値をどのように使用して、変数の値が未設定であることを意味するのかと同様です。Python の関数が None を返す場合、PROC FCMP は、None が SAS の double データ型の"値なし"を表しているのか、文字変数の"値なし"を表しているのかを認識していません。そのため、PROC FCMP は、常に返された Python の None 値を SAS の double データ型の欠損値に変換します。そのため、返された Python の None を SAS の文字変数に格納しないようにコードを記述する必要があります。

** "[1, 2, 3]"や"['a', 'b', 'c']"などの Python のリストは、変換テーブルに記載されているように、対応する型の SAS 配列に変換されます。SAS は、"[3, 1.5]"や"[4, 'x']"のような型が混在した Python リストの返却をサポートしていません。Python のリストが SAS に返された場合、SAS の配列型は、リストの中で"None"という値を持たない最初の要素によって決定されます。リストのすべての要素は、同じ SAS データ型に変換されます。

注: リスト"[1, 2.3, 4.01]"を返す場合、SAS は整数のリスト[1、2、4]を受け取ります。数字のリストを返す場合は、意図しない切り捨てが起こらないようにしなければなりません。例えば、"[float(1), 2.3, 4.01]"と返せば、先ほどの切り捨てを避けることができます。

SAS と PROC FCMP Python で Date、Datetime、Time Data を扱う

Python 関数が Python の date、datetime、time オブジェクトを PROC FCMP に返すと、そのオブジェクトは SAS の対応する date、datetime、time の値を表す数値に変換されます。Python の date、datetime、time オブジェクトを Python から SAS に渡すための追加のコードは必要ありません。

SAS の date、datetime、time の値を Python 関数に渡す場合、ユーザーは SAS の数値が適切な Python オブジェクトに正しく変換されていることを確認する必要があります。SAS では、date、datetime、time をある基準点から加算(減算)されたある単位の時間を表す数値として格納します。SAS の date の場合、数値は 1960 年 1 月 1 日からの日数を表しています。SAS の datetime の場合、数値は 1960 年 1 月 1 日の午前 0 時からの秒数を表します。SAS の time の場合、数値は午前 0 時からの秒数を表します。

これらの例では、SAS の date、datetime、または time の値が Python 関数に渡され、Python 関数は受け取った数値を適切な Python オブジェクトに変換し、フォーマットされた文字列表現を返します。

SAS の date 値を Python の date 値に変換する例:

```
proc fcmp;
length d_str $ 36;
length d_out $ 36;
declare object py(python);
submit into py;
import datetime
```

```
def get_date(indate):
  "Output: outdate"
  d = datetime.date(1960, 1, 1) + \
     datetime.timedelta(days=indate)
  return d.strftime('%m/%d/%Y')
endsubmit;
rc = py.publish();
d_in = "22Dec1983"d;
put d_in=;
d_str=put(d_in,mmddyy10.);
put d_str=;
rc = py.call("get_date", d_in);
d_out = py.results["outdate"];
put d_out=;
quit;
```

上記のステートメントは、次の結果を生成します。

```
d_in=8756
d_str=12/22/1983
d_out=12/22/1983
```

SAS の datetime 値を Python の datetime 値に変換する例:

```
proc fcmp;
length dt_str $ 36;
length dt_out $ 36;
declare object py(python);
submit into py;
import datetime
def get_datetime(indatetime):
  "Output: outdatetime"
  dt = datetime.datetime(1960, 1, 1, 0, 0, 0) + \
      datetime.timedelta(seconds=indatetime)
  return dt.strftime('%d%b%Y:%H:%M:%S')
endsubmit;
rc = py.publish();
dt_in = "22Dec1983:13:52:00"dt;
put dt_in=;
dt_str=put(dt_in,datetime20.);
put dt_str=;
rc = py.call("get_datetime", dt_in);
dt_out = py.results["outdatetime"];
put dt_out=;
quit;
```

上記のステートメントは、次の結果を生成します。

```
dt_in=756568320
dt_str=22DEC1983:13:52:00
dt_out=22Dec1983:13:52:00
```

SAS の time 値を Python の time 値に変換する例:

```
proc fcmp;
length t_str $ 36;
length t_out $ 36;
```

```
declare object py(python);
submit into py;
from datetime import date, datetime, time, timedelta
def get_time(intime):
  "Output: outtime"
  t = ( datetime.combine(date.today(), time()) + \
      timedelta(seconds=intime) ).time()
  return t.strftime('%H:%M:%S')
endsubmit;
rc = py.publish();
t_in = "17:23:45"t;
put t_in=;
t_str=put(t_in,time.);
put t_str=;
rc = py.call("get_time", t_in);
t_out = py.results["outtime"];
put t_out=;
quit;
```

上記のステートメントは、次の結果を生成します。

```
t_in=62625
t_str=17:23:45
t_out=17:23:45
```

コメント

PROC FCMP でサブミットされた Python 関数は、ハッシュ文字(#)のコメントメソッドのみをサポートしています。たとえば、""" This is my comment."""のような docstring コメントはエラーの原因となります。ここでは、PROC FCMP の Python コードを有効にコメントした例を紹介します。

```
proc fcmp;
declare object py(python);
submit into py;
def MyFunc(arg1):
   "Output: MyKey"
   #This is a comment
   ReturnVar = arg1 * 5 #This is another comment
   return ReturnVar
endsubmit;
rc = py.publish();
rc = py.call("MyFunc", 10);
x = py.results["MyKey"];
put x=;
run;
```

上記のステートメントは、次の結果を生成します。

```
x= 50
```

ステートフル Python プログラム

次の例は、ステートフルデータ、具体的にはコードを使用する Python プログラムです。yield x .“マルチスレッド環境でステートフルデータを使用する Python オブジェクトの操作”で説明したように、ステートフルデータを使用する Python プログラムをマルチスレッド SAS 環境で実行すると、結果が不確定になる可能性があります。

```
function py_pq3(price);
declare object py_pq3_module(python);
rc=0;
submit into py_pq3_module;
def my_generator():
   for x in range(5):
      yield x

shared_gen = my_generator()

def get_sum_share():
   "Output: sum_share"
   sum = 0
   for x in shared_gen:
      sum += x
   return sum
```

yield キーワードを含む関数は、グローバルレベルで宣言されたジェネレーター関数です。関数が呼び出された後、関数内のローカル状態は保持されます。マルチスレッド SAS 環境では、複数のスレッドが複数の対応する Python インタープリターセッション(スレッドごとに 1 つ)を作成して、グローバルジェネレーター関数を呼び出します。これにより、複数のスレッドが同じグローバルストレージに書き込むことになり、予期しない結果が生じる可能性があります。この動作は、yield ステートメントに限ったことではありません。この次の例では、クラス変数 Counter はグローバルレベルでインスタンス化されます。

```
class Counter(object):
   def __init__(self):
      self.internal_counter = 0
   def increment(self):
      self.internal_counter += 1
   @property
   def count(self):
      return self.internal_counter

myCounter = Counter()

def click():
   "Output: count"
   myCounter.increment()
   return myCounter.count
```

11

Python オブジェクトの言語要素

ディクショナリ **263**
APPEND メソッド 263
CALL メソッド 265
CALL CLEAR メソッド 267
DECLARE ステートメント: Python オブジェクト 268
INFILE メソッド 269
PUBLISH メソッド 270
RESULTS ディクショナリ メソッド 271
RTINFILE メソッド 273
SUBMIT INTO ステートメント 274

ディクショナリ

APPEND メソッド

実行時に一行の Python コードを Python オブジェクトに追加します。

適用対象: Python オブジェクト

参照項目: 詳細については、Python オブジェクトの使用を参照してください。

構文

rc=*object-reference*.**APPEND**("*Python-code*");

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
Python オブジェクトの参照名を指定します。

"*Python-code*"
Python オブジェクトに追加される Python コードの行を指定します。コードの行は二重引用符で囲み、Python の標準構文とインデントルールに従わなければなりません。

詳細

APPEND メソッドは、Python オブジェクト内の既存の Python コードに一行のコードを追加します。オブジェクトに Python コードが格納されていない場合、APPEND メソッドは Python コードの最初の行を作成します。複数の APPEND メソッド文を使用することで、Python オブジェクトに複数行のコードを追加することができます。追加が成功しても、Python コードの構文が有効かどうかはチェックされません。PUBLISH メソッドを使用して、Python コードの構文が正しいかどうかチェックします。APPEND メソッドと他のコードサブミットメソッドとの違いについては、“サブミットメソッドの選択”を参照してください。

例: Python コードの Python オブジェクトへの追加

```
proc fcmp;
declare object py(python);
submit into py;
def MyPyFunc(arg1):
   "Output: MyKey"
   myvar = arg1 * 10
endsubmit;
rc = py.append("    return myvar");
rc = py.publish();
rc = py.call("MyPyFunc", 5);
a = py.results["MyKey"];
put a=;
run;
```

上記のステートメントは、次の結果を生成します。

```
a= 50
```

CALL メソッド

Python オブジェクトに格納された Python 関数を実行します。

適用対象: Python オブジェクト

注: CAS サーバーで CALL メソッドを実行してエラーが発生した場合、エラーメッセージは SAS のログではなく app.cas.actions.FCMPACT のロガーに書き込まれます。

参照項目: 詳細については、Python オブジェクトの使用を参照してください。
CAS ログの詳細については、CAS サーバーのロガーを参照してください。

構文

rc=*object-reference*.**CALL**(<*dictionary-reference*>, "*python-function-name*" <, *python-argument-1*> <, *python-argument-2*, ...>);

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
Python オブジェクトの参照名を指定します。

"*python-function-name*"
Python オブジェクトに格納されている Python 関数名を指定します。関数名は二重引用符(" ")で囲む必要があります。

オプション引数

dictionary-reference
ディクショナリオブジェクトの参照名を指定します。

python-argument
Python の関数が必要とする引数を指定します(もしあれば)。

詳細

CALL メソッドは、Python オブジェクト内に格納されているパブリッシュされている Python 関数を実行するために使用されます。Python オブジェクトではなく、Python 関数の名前を指定し、二重引用で囲む必要があります。Python 関数名は、大文字と小文字を区別します。Python 関数が引数を必要とする場合は、関数名の後にカンマ(,)で区切って指定する必要があります。

CALL メソッドからの出力はディクショナリオブジェクトに格納されています。オプションのディクショナリオブジェクトを指定して、関数から返された出力を格納できます。ディクショナリオブジェクトが指定されていない場合、結果は Python オブジェクトのメンバーディクショナリに格納され、関数が呼び出された後に、RESULTS ディクショナリを使って返されます。

出力値は、Python 関数出力文字列で指定された key-name を持つディクショナリオブジェクトに索引を付けて SAS で返されます。次の例では、CALL メソッドの両方の形式について詳細を説明しています。

例

例 1: Python オブジェクトからの Python 関数の呼び出しと Python オブジェクトのメンバーディクショナリの使用

```
proc fcmp;
declare object py(python);
x = 5; FCMP_Result = .; rc = 0;

submit into py;
def MyPyFunction(Arg1):
   "Output: MyOutputKey"
   MyPyResult = Arg1 * 10
   return MyPyResult
endsubmit;

rc = py.publish();
rc = py.call("MyPyFunction", x);
FCMP_Result = py.results["MyOutputKey"];
put FCMP_Result=;
run;
```

上記のステートメントは、次の結果を生成します。

```
FCMP_Result=50
```

例 2: Python オブジェクトからの Python 関数の呼び出しとディクショナリオブジェクトの指定

```
proc fcmp;
declare object py(python);
declare dictionary d;
x = 5; FCMP_Result = .; rc = 0;

submit into py;
def MyPyFunction(Arg1):
   "Output: MyOutputKey"
   MyPyResult = Arg1 * 10
   return MyPyResult
endsubmit;

rc = py.publish();
```

```
rc = py.call(d, "MyPyFunction", x);
FCMP_Result = d["MyOutputKey"];
put FCMP_Result=;
run;
```

上記のステートメントは、次の結果を生成します。

```
Result=50
```

CALL CLEAR メソッド

サブミットバッファーと結果ディクショナリをクリアして、既存のオブジェクトに新しいコードをパブリッシュできるようにします。

適用対象: Python オブジェクト

参照項目: 詳細については、Python オブジェクトの使用を参照してください。

構文

CALL *object-reference*.**CLEAR**();

必須引数

object-reference
Python オブジェクトの参照名を指定します。

詳細

CALL CLEAR メソッドは、PUBLISH メソッドを使用してサブミットバッファーに格納されている Python のソースコードと RESULTS ディクショナリに格納されているすべての結果をクリアします。Python オブジェクトは残り、新しい Python ソースコードのサブミットに使用できます。サブミットバッファーのクリアに成功すると、ユーザーは既存の Python オブジェクトに新しい Python ソースコードをサブミットし、そのソースコードをサブミットバッファーにパブリッシュすることができます。

例: Python オブジェクトのクリア

```
proc fcmp;
  declare object py(python);
  rc = py.publish();
  call py.clear();
```

```
run;
```

CALL CLEAR メソッドは出力を返しません。

DECLARE ステートメント: Python オブジェクト

Python オブジェクトを宣言します。

別名: DCL

適用対象: Python オブジェクト

参照項目: 詳細については、Python オブジェクトの使用を参照してください。

構文

DECLAREOBJECT*object-reference*(**PYTHON**<("*module-name*")>);

必須引数

object-reference
Python オブジェクトの参照名を指定します。

オプション引数

"*module-name*"
Python オブジェクト内に格納されているモジュール名を指定します。

注: これは関数の名前ではなく、関数を作成しません。

詳細

DECLARE ステートメントは、Python オブジェクトを作成し、Python モジュールに名前を付けるために使用されます。Python オブジェクトは、PROC FCMP プログラムの中でのみ宣言できます。

例

```
proc fcmp;
  declare object py(python("MyPyModule"));
  declare object py2(python);
run;
```

Python オブジェクトを宣言しても、返される出力はありません。

INFILE メソッド

解析時に Python ソースコードをファイルから Python オブジェクトに読み込みます。

適用対象: Python オブジェクト

参照項目: 詳細については、Python オブジェクトの使用を参照してください。

構文

rc=*object-reference*.**INFILE**("*file-path*");

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
Python オブジェクトの参照名を指定します。

"*file-path*"
ローカルの Python ファイルのファイルパスを指定します。ファイルパス値は二重引用符(" ")で囲む必要があります。

詳細

INFILE メソッドは、Python ソースコードをローカルファイルから Python オブジェクトに読み込みます。ファイル内のソースコードは、Python の構文とインデントルールに従う必要があります。ソースコードは、Python の関数定義ステートメントの後に出力文字列 (256 ページ)を付けて変更する必要があります。INFILE メソッドと他のコードサブミットメソッドとの違いについては、“サブミットメソッドの選択”を参照してください。

例: INFILE メソッドで Python ソースコードを Python オブジェクトに読み込む

この例では、"MyPyFunction.py"というファイルから Python ソースコードを読み込みます。

```
def MyPyFunc(arg1):
  "Output: MyOutputKey"
  PyResult = arg1 * 10
  return PyResult
```

このファイルのファイルパスを使って、Python 関数を PROC FCMP からサブミットして呼び出すことができます。

```
proc fcmp;
  declare object py(python);
  rc = py.infile("Your-File-Path/MyPyFunction.py");
  put rc=;
  rc = py.publish();
  rc = py.call("MyPyFunc", 5);
  Result = py.results["MyOutputKey"];
  put Result=;
run;
```

上記のステートメントは、次の結果を生成します。

```
rc=0
Result=50
```

PUBLISH メソッド

PROC FCMP で Python のソースコードを Python インタープリターにサブミットします。

適用対象: Python オブジェクト

参照項目: 詳細については、Python オブジェクトの使用を参照してください。

構文

rc=*object-reference*.**PUBLISH**();

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
Python オブジェクトの参照名を指定します。

詳細

PUBLISH メソッドは、Python オブジェクトに格納されている全ての Python ソースコードを Python インタープリターにサブミットします。Python ソースコードは、これらのメソッドやステートメントを使って Python オブジェクトに読み込むことができます。

- APPEND メソッド

- INFILE メソッド
- RTINFILE メソッド
- SUBMIT INTO ステートメント

CALL メソッドを Python オブジェクトに使って、Python ソースコードを実行する前に、PUBLISH メソッドを実行する必要があります。Python コードの構文エラーにより、PUBLISH メソッドが失敗し、エラーをログに返します。

例: PUBLISH メソッドを使って Python のソースコードを Python インタープリターにサブミットする

```
proc fcmp;
declare object py(python);
rc = 0;

submit into py;
def MyPyFunction(Arg1):
   "Output: MyOutputKey"
   MyPyResult = Arg1 * 10
   return MyPyResult
endsubmit;

rc = py.publish();
run;
```

PUBLISH メソッドが返す出力はありません。

RESULTS ディクショナリ メソッド

メンバー変数のディクショナリを使った Python 関数呼び出しの結果を返します。

適用対象: Python オブジェクト

参照項目: 詳細については、Python オブジェクトの使用を参照してください。

構文

形式 1: *data-variable*=*object-reference*.**RESULTS**["*output-key*"];

形式 2: *rc*=*object-reference*.**RESULTS**.**CLEAR**();

必須引数

data-variable
RESULTS ディクショナリから指定されたキーの出力のコピーを格納する変数を指定します。

object-reference
Python オブジェクトの参照名を指定します。

"*output-key*"
Python 関数の戻り値の出力キー名を指定します。

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

詳細

RESULTS ディクショナリは、ディクショナリオブジェクトを宣言せずに、呼び出された Python 関数からの出力にアクセスするために使用されます。ディクショナリオブジェクトが"CALL メソッド"に指定されていない場合、CALL メソッドは RESULTS メンバーディクショナリに出力を格納します。結果にアクセスするには、メンバーディクショナリから出力キー (256 ページ)を指定することで、必要な戻り値を得ることができます。

RESULTS メンバーディクショナリに格納されている出力をクリアするには、構文の形式 2 に示すディクショナリ"CLEAR メソッド" (227 ページ)を使用します。RESULTS ディクショナリに.CLEAR を付加すると、ディクショナリからすべてのデータアイテムが削除されます。

注: CALL メソッドは、RESULTS ディクショナリの同名のデータアイテムを 自動的に上書きします。同じ関数を実行して新しい結果を返すには、RESULTS ディクショナリのクリアは必要ありません。

例: RESULTS メソッドを使って Python 関数から結果を返す

```
proc fcmp;
  declare object py(python);
  x = 10; myresult = .; rc = 0;
  submit into py;
  def MyFunc(arg1):
    "Output: MyPyResult"
    result = arg1 * 2
    return result
  endsubmit;
  rc = py.publish();
  rc = py.call("MyFunc", x);
  myresult = py.results["MyPyResult"];
  put myresult=;
quit;
```

上記のステートメントは、次の結果を生成します。

```
myresult= 20
```

RTINFILE メソッド

実行時に Python ソースコードをファイルから Python オブジェクトに読み込みます。

適用対象: Python オブジェクト

参照項目: 詳細については、Python オブジェクトの使用を参照してください。

構文

rc=*object-reference*.**RTINFILE**("*file-path*");

必須引数

rc
メソッドが成功したか失敗したかを示します。0(ゼロ)のリターンコードは成功を表し、ゼロ以外の値は失敗を表します。

object-reference
Python オブジェクトの参照名を指定します。

file-path
サーバー上の Python ファイルへのファイルパスを指定します。ファイルパス値は二重引用符で囲む必要があります。

詳細

RTINFILE メソッドは、クライアント/サーバー環境で、サーバー上のファイルから Python ソースコードを読み込むために使用されます。ソースコードは、Python の構文とインデントルールに従う必要があります。ソースコードは、PROC FCMP での Python 関数定義の後に出力文字列を付けて変更する必要があります。RTINFILE メソッドと他のコードサブミットメソッドの違いについては"サブミットメソッドの選択"を参照してください。

例: サーバー上のファイルからの Python ソースコードの読み込み

この例では、"MyPyFunction.py"というファイルから Python ソースコードを読み込みます。

```
def MyPyFunc(arg1):
```

```
  "Output: MyOutputKey"
  PyResult = arg1 * 10
  return PyResult
```

このファイルのファイルパスを使って、Python 関数を PROC FCMP からサブミットして呼び出すことができます。

```
proc fcmp;
  declare object py(python);
  rc = py.rtinfile("Your-File-Path/MyPyFunction.py");
  put rc=;
  rc = py.publish();
  rc = py.call("MyPyFunc", 5);
  Result = py.results["MyOutputKey"];
  put Result=;
run;
```

上記のステートメントは、次の結果を生成します。

```
rc=0
Result=50
```

SUBMIT INTO ステートメント

解析時に Python ソースコードを Python オブジェクトに挿入します。

適用対象: Python オブジェクト

参照項目: 詳細については、Python オブジェクトの使用を参照してください。

構文

形式 1: **SUBMIT INTO** *object-reference*; *...python-code...* **ENDSUBMIT;**

形式 2: **SUBMIT INTO** *object-reference*("*file-path*");

必須引数

object-reference
Python オブジェクトの参照名を指定します。

"file-path"
Python のソースコードを含むファイルのファイルパスを指定します。

詳細

SUBMIT INTO ステートメントは、Python コードが埋め込まれている、または Python コードを含むファイルのある Python オブジェクトに Python ソースコード

を挿入するために使用されます。どちらの方法でも、Python コードは、PROC FCMP で実行する関数定義の後に出力文字列を付けて変更する必要があります。埋め込みサブミットブロックは、ユーザーが SAS プログラムに Python 関数を埋め込むためのソースコードサブミットブロックを作成します。埋め込みサブミットブロックに書かれた Python 関数を呼び出して、結果を返すことができます。サブミットブロックを終了するには、ENDSUBMIT ステートメントを使用します。SUBMIT INTO ステートメントは、第 2 形式を使用し、INFILE メソッドと同様の *file-path* 引数を指定することで、Python ファイルのサブミットもサポートしています。

注: "*file-path*"形式を使用している場合は、ENDSUBMIT ステートメントを使用しないでください。ファイルパスを指定した後に ENDSUBMIT ステートメントを使用すると、エラーが発生します。

埋め込みコードをサブミットする際、SUBMIT INTO ステートメントと ENDSUBMIT ステートメント間の行はホワイトスペースの有無を区別します。行は一度に一つずつ読み込まれ、指定された Python オブジェクトのサブミットバッファーに順次格納されます。埋め込み Python コードブロックを使用する場合、SUBMIT INTO ステートメントおよび ENDSUBMIT ステートメントをインデントしないでください。これらのステートメントは、SAS プログラムエディターの最初(一番左)の位置から始める必要があります。Python の構文やインデントに関する標準的なルールが、埋め込みサブミットブロックに書かれたソースコードに適用されます。埋め込みサブミットブロックは Python の構文エラーを検出しません。PUBLISH メソッドを使って、Python オブジェクトが Python インタープリターにサブミットされると、エラーが検出されます。SUBMIT INTO ステートメントと他のコードサブミットメソッドの違いについては、“サブミットメソッドの選択” (254 ページ)を参照してください。

例

例 1: 埋め込みサブミットブロック形式を使って Python ソースコードをサブミットする

```
proc fcmp;
declare object py(python);
submit into py;
def MyPyFunc(Arg1):
   "Output: MyOutputKey"
   MyPyResult = Arg1 * 10
   return MyPyResult
endsubmit;
run;
```

これらのステートメントによって返される出力はありません。

例 2: ファイルサブミット形式を使って Python ソースコードをサブミットする

この例では、"MyPyFunction.py"というファイルから同じ Python ソースコードを読み込みます。ファイルにはこのようなコードが含まれています。

```
def MyPyFunc(Arg1):
```

```
"Output: MyOutputKey"
MyPyResult = Arg1 * 10
return MyPyResult
```

```
proc fcmp;
  declare object py(python);
  submit into py("Your-File-Path/MyPyFunction.py");
run;
```

これらのステートメントによって返される出力はありません。