artisoc Cloud

artisoc4 と artisoc Cloud の違い

株式会社 構造計画研究所

mas-support@kke.co.jp

目次

関数対応表

artisoc4 と artisoc Cloud の関数の対応についてまとめています。

引数など細かい仕様が異なる場合がありますので、詳しくは関数仕様を参照してください。

数値計算

artisoc4	artisoc Cloud
Abs	abs
Atn	atan
Cos	cos
DegreeToRad	radians
Exp	exp
Int	int
Log	log
NormInv	normalvariate
Pi	pi

RadToDegree	degrees
Rnd	rand
Round	round
SetRandomSeed	(なし)
Sin	sin
Sinh	sinh
Sqr	sqrt
Tan	tan
Tanh	tanh

文字列操作

artisoc4	artisoc Cloud
CountToken	len
GetToken	(リストのインデックス指定で記述)
Instr	in
Left	(リストのスライスで記述)
Len	len
Mid	(リストのスライスで記述)
Replace	replace
Right	(リストのスライスで記述)
StrComp	演算子==で記述
Trim	strip

データ型変換

artisoc4	artisoc Cloud
Cbool	bool
CDbl	float
Cint	int
Clong	int
Cstr	str

エージェント操作（生成・削除）

artisoc4	artisoc Cloud
CreateAgt	create_agt
DelAgt	del_agt
KillAgt	kill_agt
TerminateAgt	（なし）

エージェント操作（その他）

artisoc4	artisoc Cloud
GetHistory	get_history
SpecifyAgttype	specify_agttype
SpecifyKillAgt	specify_kill_agt

空間エージェント関数（移動）

artisoc4	artisoc Cloud
Forward	forward
ForwardDirectionCell	forward_direction_sqgrid
MoveToCenter	move_center
MoveToSpaceOwnCell	move_space_around_own_sqgrid
MoveToSpaceAgtSetCell MoveToSpacePositionCell	move_space_around_position_sqgrid
Pursue	pursue
Turn	turn
TurnAgt	turn_agt

空間エージェント関数（空間操作）

artisoc4	artisoc Cloud
GetHeightSpaceOwn	get_height_space_own
GetWightSpaceOwn	get_wight_space_own
ReverseDirectionCell	reverse_direction_sqgrid

空間エージェント関数（エージェント集合生成）

artisoc4	artisoc Cloud
MakeAllAgtsetAroundOwn MakeOneAgtsetAroundOwn	make_agtset_around_own
MakeAllAgtsetAroundOwnCell MakeOneAgtsetAroundOwncell	make_agtset_around_own_sqgrid

エージェント集合操作（基本操作）

artisoc4	artisoc Cloud
AddAgt	add
DelAgtSet2 RemoveAgt	discard
ClearAgtSet	clear
CountAgtSet	count_agtset

エージェント集合操作（集合演算）

artisoc4	artisoc Cloud
MakeCommonAgtset	intersection
MakeDiffAgtset	symmetric_difference
MergeAgtSet	union
DelAgtset	difference

エージェント集合操作（配置）

artisoc4	artioc Cloud
RandomPutAgtset	random_put_agtset
RandomPutAgtsetCell	random_put_agtset_sqgrid

エージェント集合操作（生成）

artisoc4	artisoc Cloud
MakeAgtset MakeAgtsetSpace	make_agtset
MakeAllAgtsetAroundPosition MakeOneAgtsetAroundPosition	make_agtset_around_position
MakeAllAgtsetAroundPositionCell MakeOneAgtsetAroundPositionCell	make_agtset_around_position_sqgrid

エージェント集合操作（その他）

artisoc4	artisoc Cloud
SortAgtset	make_agtlist sort_agtlist

空間操作

artisoc4	artisoc Cloud
GetDirection	get_direction
GetHeightSpace	get_height_space
GetHeightSpaceOwn	get_height_space_own
GetLayerSpace	get_layer_space
GetRideSpace	get_ride_space
GetWidthSpace	get_width_space
GetWidthSpaceOwn	get_width_space_own
MeasureDistance	measuer_distance
SpecifyLoop	specify_loop

ファイル入出力

artisoc4	artisoc Cloud
CloseFile	close
OpenFile	open
ReadFile	read
WriteFile	write

その他

artisoc4	artisoc Cloud
ExitSimulation	exit_simulation
GetCountStep	count_step
Gradation	gradation
HSV	hsv
PauseSimulation	pause_simulation
Print Println	print
RGB	rgb

基本的な記述ルール

artisoc4 と artisoc Cloud の基本的な記述ルールの違いについて説明します。

なお、artisoc Cloud の基本的な記述ルールは Python のものと共通ですので、詳細な点は Python 公式ドキュメントなどを参照してください。

大文字と小文字

artisoc4 では大文字と小文字を区別しません。一方、artisoc Cloud では大文字と小文字を区別しますので注意してください。

artisoc4

```
PrintLn(My.X)  //エラーにならない
PrintLn(my.x)  //小文字でもエラーにならない
```

artisoc Cloud

```
print(self.x)  # エラーにならない
print(self.X)  # エージェント座標 x は小文字で定義されているので、大文字だとエラー
```

コメントアウト

artisoc4

`//`で定義します。

```
//これはコメントです
```

artisoc Cloud

`#`で定義します。

```
#これはコメントです
```

改行とインデント

artisoc4

artisoc4 では、改行とインデントについてのルールはありません。つまり、改行とインデントはコードを見やすくするために挿入するものであって、改行とインデントがなくともコードは問題なく動作します。

```
//改行とインデントを入れた見やすいコード
Agt_Init{
  My.X = 3
  My.Y = 3
}
//改行やインデントがないコードでも問題なく動く
Agt_Init{
MY.X = 3  My.Y = 3
}
```

artisoc Cloud

artisoc Cloud では、改行とインデントを正しく用いないとエラーになります。

```
# 正しいコード
def agt_init(self):
    self.x = 3
    self.y = 3
# エラーになるコード
def agt_init(self):
    self.x = 3
    self.y = 3  # インデントが揃っていないためエラー
```

演算子

四則演算の`+, -, *, /`、代入の`=`など、おおよそ演算子は共通です。

主な違いとしては、「値が等しくない」ことを表す演算子`<>`が artisoc Cloud では使えません。代わりに`!=`を用います。

artisoc4

```
If My.X <> 3 Then
    [処理]
End If
```

artisoc Cloud

```
if self.x != 3:
    [処理]
```

また、artisoc Cloud では変数に値を加える演算子`+=`を用いることができます。同様に、`-=, *=, /=`を用いることもできます。

artisoc4

```
Universe.hensuu = Universe.hensuu + 1  //hensuu に 1 を加える
```

artisoc Cloud

```
Universe.hensuu = Universe.hensuu + 1  # hensuu に 1 を加える
Universe.hensuu += 1  # このように書いても OK
```

変数の型

artisoc4

artisoc4 では、変数の型はツリーで指定するか、ルールエディタ内で `Dim` を用いて定義します。定義した型に合わない値を代入しようとするとエラーになります。

```
//整数型変数 i を定義し、値を代入
Dim i as integer
i = 3
i = "hoge"  //整数型変数に文字列型の値を代入しようとするとエラー
```

artisoc Cloud

artisoc Cloud では変数の型定義は必要ありません。変数の型は値が代入されたときに決定され、その変数に別の型の値を上書きすることもできます。

```
i = 3  # 変数 i を定義し、整数型の値 3 を代入
i = "hoge"  # 変数 i に文字列型の値"hoge"を上書きすることができる
```

変数の初期化

artisoc4

変数の値は変数の型に応じて決定されます。例えば、数値型の変数であれば初期値は0、エージェント集合型の変数であれば空の集合となります。

ツリー上で作成した変数であれば、「初期値設定」機能を用いて変数の初期値を変更することも可能です。

```
Univ_Init{
  one = CreateAgt(Universe.hiroba.hito)
  AddAgt(Universe.people, one)  //ツリー上で作成した Universe 変数 people（初期値は空）にエージェントを追加
}
```

artisoc Cloud

ツリー上で作成する変数の場合、初期値は整数の 0 となります。別の初期値を与えたい場合は、univ_init や agt_init で値を代入する必要があります。

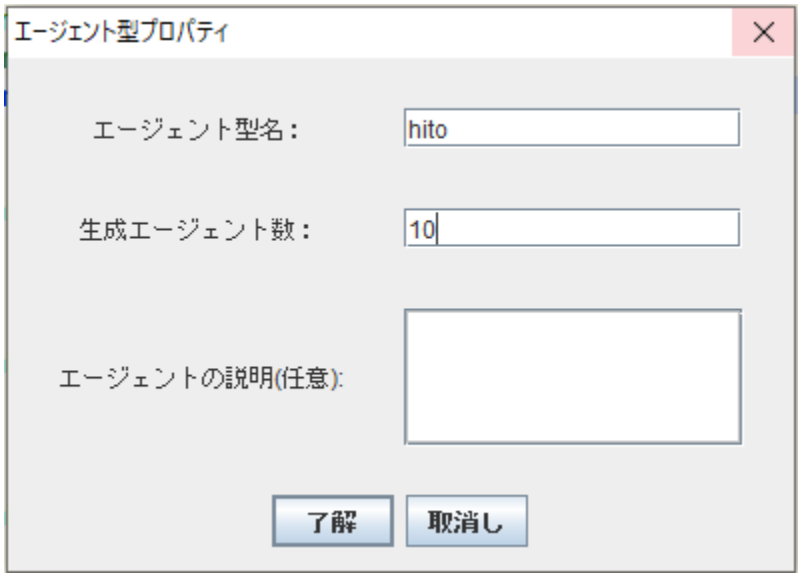
たとえば、エージェント集合型の変数を Universe 変数として作成する場合は、必ず univ_init で初期化します。

```
def univ_init(self):

    Universe.people = set()  # Universe 変数 people を空の集合型変数として初期化

    one = create_agt(Universe.hiroba.hito)
    Universe.people.add(one)  # 空の集合型変数である people にエージェント one を追加
```

エージェントの初期生成

artisoc4

artisoc4 では、ツリーからエージェントのプロパティを操作することでエージェントの初期生成を行うことができます。下図では、シミュレーション開始時にエージェントを 10 体生成しています。



artisoc Cloud

artisoc Cloud では、ツリー上からエージェントを生成することはできません。エージェントを生成するときは必ずルールエディタ内で create_agt 関数を用います。シミュレーション開始時に生成したい場合は、univ_init 内に記述します。

```
def univ_init(self):
    create_agt(Universe.hiroba.hito, num=10)  # シミュレーション開始時に hito エージェントを 10 体生成
```

if 文・for 文

代表的な制御構文である if 文と for 文について、artisoc4 と artisoc Cloud の違いを解説します。

if 文

artisoc4

artisoc4 では if 文は以下のように記述します。

```
If [条件文 1] Then
    [処理 1]
ElseIf [条件文 2] Then
    [処理 2]
Else
    [処理 3]
End If
```

※ElseIf, Else は省略可。ElseIf は複数書くことも可。

artisoc Cloud

artisoc Cloud では、if 文は以下のように記述します。処理はインデント内に記述することに注意します。

```
if [条件文 1]:
    [処理 1]
elif [条件文 2]:
```

```
    [処理 2]
else:
    [処理 3]
```

※elif, else は省略可。elif は複数書くことも可。

for 文

繰り返し処理

例として、n 回の繰り返し処理を考えます。

artisoc4

artisoc4 では、以下のように記述します。

```
For i = 0 to n - 1
    [処理]
Next i
```

artisoc Cloud

artisoc Cloud では、以下のように `range` 関数を用いて記述します。処理をインデント内に記述することに注意します。

```
for i in range(n):
    [処理]
```

集合の各要素に対する処理

例として、エージェント集合 people の各要素 one に対する処理を考えます。

artisoc4

artisoc4 では、下記のように For Each 文を用います。

```
For Each one in people
    [処理]
Next one
```

artisoc Cloud

artisoc Cloud では、以下のように記述します。

```
for one in people:
    [処理]
```

集合型

artisoc4 と artisoc Cloud における集合型変数の扱いについて解説します。

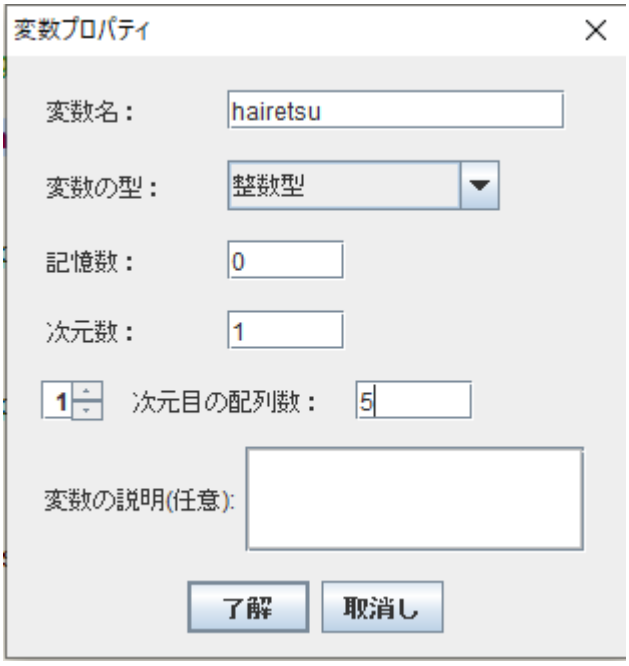
集合型変数とはデータの集まりを表現する変数です。artisoc4 では**配列**や**エージェント集合型**の変数、artisoc Cloud では**リスト型**や**セット型**の変数を用いて実現します。

配列とリスト

artisoc4 での配列は、artisoc Cloud においてはリスト型変数を用いることでほぼ同様のことが実現できます。

artisoc4

artisoc4 において配列変数を用いるためには、配列のサイズをツリー上であらかじめ定義します。下図の例では、長さ 5 の 1 次元配列を定義しています。



配列変数へアクセスするためには、`()`を用いて要素番号を指定します。

```
Universe.hairetsu(3) = 5  //配列変数の 3 番目の要素に 5 を代入
```

artisoc Cloud

artisoc Cloud においては、artisoc4 での配列に正確に対応するデータ型は存在しません。その代わり、**リスト型**の変数を用いることができます。

リスト型の変数を定義するには、下のように`[]`を用います。

```
Universe.list1 = [1, 2, 3, 4, 5]  # 長さが 5 のリストを定義
```

リスト型の変数にアクセスするには、`[]`を用いて要素番号を指定します。

```
Universe.list1[3] = 10  # リストの 3 番目の要素に 10 を代入
print (Universe.list1)  # --> [1, 2, 3, 10, 5] （3 番目の要素に 10 が入っている。
要素番号は 0 から数えることに注意）
```

エージェント集合型とセット型・リスト型

artisoc4 でのエージェント集合型変数は、artisoc Cloud では**セット型**もしくは**リスト型**の変数を用いて表現します。以下の典型的な 2 つのケースに分けて説明します。

- 関数を用いてエージェント集合を取得するケース
- 空のエージェント集合に要素を追加していくケース

関数を用いてエージェント集合を取得するケース

たとえば、関数を用いて周囲にいるエージェントを取得するケースを考えます。

artisoc4

たとえば `MakeAllAgtsetAroundOwn` 関数を用いることで、周囲にいるエージェントをエージェント集合型の変数として取得できます。このとき、エージェント集合型変数を事前に定義しておく必要があります。

```
Dim neighbors as Agtset
MakeAllAgtsetAroundOwn(neighbors, 2, False)  //自分から距離 2 以内にいるエージェントをエージェント集合型変数 neighbors に格納する
```

取得したエージェント集合から要素を取得するには、GetAgt 関数を用いて要素番号を指定します。ランダムに 1 つの要素を取得する場合には、たとえば以下のようにします。

```
one = GetAgt(neighbors, int(rnd() * CountAgtset(neighbors)))  //ランダムな要素番号を生成することで、ランダムに要素を取得
```

artisoc Cloud

周囲にいるエージェントを取得するためには、たとえば `make_agtset_around_own` 関数を用います。このとき、エージェント集合型変数を事前に定義しておく必要はありません。また、関数名を `self.` に続けて記述すること、エージェント集合は関数の戻り値として与えられることに注意します。

```
neighbors = self.make_agtset_around_own(2, False)
```

このとき、エージェント集合 neighbors は**セット型**の変数です。セット型は要素の順番が定義されず、したがって要素番号を指定して要素を取得することができないことに注意する必要があります。

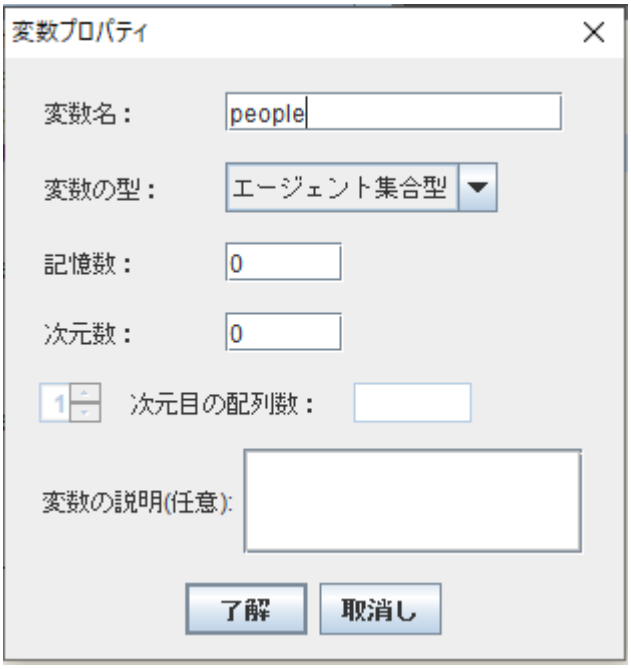
ランダムに 1 つの要素を取得する場合には、以下のように **randchoice** 関数を用います。

```
one = randchoice(neighbors)
```

空のエージェント集合に要素を追加していくケース

artisoc4

下図のように、ツリー上でエージェント集合型変数 people を定義する場合を考えます。



このとき、変数の初期値は空のエージェント集合です。AddAgt 関数を用いることで、ここにエージェントを追加していくことができます。

```
taro = create_agt(Universe.hiroba.hito)
AddAgt(Universe.people, taro)  //エージェント集合 people にエージェント taro を追加

hanako = CreateAgt(Universe.hiroba.hito)
AddAgt(Universe.people, hanako)  //エージェント集合 people にエージェント hanako を
追加
```

この時点で、people には 0 番目の要素に taro、1 番目の要素に hanako が格納されています。（要素番号は 0 番から数えることに注意してください）

エージェントを取り出すためには、GetAgt 関数で位置を指定して取得します。

```
one = GetAgt(Universe.people, 1)  //1 番目の要素（hanako）を取得
```

artisoc Cloud

artisoc Cloud で同じようなルールを実現するためには、**リスト型**または**セット型**の変数を用います。リスト型は要素の順番が定義されますが、セット型では定義されません。

まずは、**リスト型**の変数を用いる場合を見てみます。

ツリー上で Universe 変数 people_list を定義したとき、初期値は整数の 0 です。これをリスト型として用いるためには、univ_init 内で初期化する必要があります。`list()`と記述することで空のリストを生成できますので、これを univ_init 内で代入します。

```
def univ_init(self):
  Universe.people_list = list()  # 変数 people_list を空のリストとして初期化
```

リスト型変数に要素を追加するには **append** 関数を用います。

```
taro = create_agt(Universe.hiroba.hito)
Universe.people_list.append(taro)  # people に taro を追加

hanako = create_agt(Universe.hiroba.hito)
Universe.people_list.append(hanako)  # people に hanako を追加
```

この時点で、people には 0 番目の要素に taro、1 番目の要素に hanako が格納されています。（要素番号は 0 番から数えることに注意してください）

エージェントを取り出すためには、`[]`を用いて位置を指定します。

```
one = Universe.people_list[1]  # 1 番目の要素（hanako）を取得
```

続いて、**セット型**の変数を用いる場合を見てみます。

ツリー上で universe 変数 people_set を定義します。このとき初期値は整数の 0 ですので、univ_init で空のセットとして初期化します。空のセットを生成するには `set()`と記述します。

```
def univ_init(self):
    Universe.people_set = set()  # 変数 people_set を空のセットとして初期化
```

セット型変数に要素を追加するには **add** 関数を用います。

```
taro = create_agt(Universe.hiroba.hito)
Universe.people_set.add(taro)  # people_set に taro を追加

hanako = create_agt(Universe.hiroba.hito)
Universe.people_set.add(hanako)  # people_set に hanako を追加
```

セット型の変数では、要素の順番が定義されません。したがって、位置を指定して要素を取り出すことができないことに注意が必要です。

主な操作のまとめ

以下、people を集合型変数（エージェント集合型／セット型／リスト型）、taro をエージェントとして、基本的な操作についてまとめます。

要素の追加

artisoc4（エージェント集合型）

```
AddAgt(people, taro)
```

artisoc Cloud（セット型）

```
people.add(taro)
```

artisoc Cloud（リスト型）

```
people.append(taro)
```

要素の取得

artisoc4（エージェント集合型）

```
one = GetAgt(people, 1)  # 1 番目の要素を取得
```

artisoc Cloud（セット型）

```
one = randchoice(people)  # ランダムに 1 つの要素を取得
```

※要素の順番が定義されないため、位置を指定して取得することはできない

artisoc Cloud（リスト型）

```
one = people[1]  # 1 番目の要素を取得
```

要素の削除

artisoc4（エージェント集合型）

```
RemoveAgt(people, taro)
```

artisoc Cloud（セット型）

```
people.discard(taro)
people.remove(taro)
```

※remove の場合、集合に存在しない値を指定するとエラーになる

artisoc Cloud（リスト型）

```
people.remove(taro)
```

※リストに複数の同じ要素が存在する場合、最初の要素を削除する

要素の探索

artisoc4

GetAgtEntry 関数を用います。要素が含まれている場合、要素番号のうち最も小さい値を整数で返します。含まれていない場合、-1 を返します。

```
//people に taro が含まれている場合、処理を行う
If GetAgtEntry(people, taro) >= 0 Then
  [処理]
End If
```

artisoc Cloud(セット型／リスト型)

演算子 in を用います。要素が含まれているかどうかを True or False で返します。

```
//people に taro が含まれている場合、処理を行う
if taro in people:
    [処理]
```

集合のクリア

artisoc4（エージェント集合型）

```
ClearAgtset(people)
```

artisoc Cloud（セット型／リスト型）

```
people.clear()
```

セット型からリスト型への変換

artisoc Cloud には、セット型からリスト型に変換する **make_agtlist** 関数が用意されています。

セット型からリスト型への変換にはこの関数を利用します。このとき、エージェントの属性値によって並べ替えをすることができます。

```
# セット型の変数 people_set を属性 tall の昇順に並べたリストを取得する
people_list = make_agtlist(people_set, key="tall")
```

単にリスト型に変換するだけであれば、**list** 関数でも可能です。

```
# セット型の変数 people_set をリスト型に変換する
people_list = list(people_set)
```

セット型とリスト型の使い分け

セット型とリスト型には、主に以下のような違いがあります。

- 要素の順番 - セット型は持たず、リスト型は持つ
- 要素の重複 - セット型は要素の重複を許さず、リスト型は持つ
- 要素の探索- セット型はリスト型に比べて要素の探索が早い

セット型の長所の１つは、要素の探索が早いことです。要素の探索はセット型もリスト型も `in` を用いて同じ方法で行いますが、リスト型に比べてセット型のほうが素早く終わります。

```
if hanako in people:
  [hanako が people に含まれる場合の処理]
```

したがって、要素の探索を頻繁に行う場合はセット型を、要素に順番を持たせたい場合はリスト型を用いるとよいでしょう。

ユーザ定義関数

ユーザ定義関数の作成および呼び出しについて、artisoc4 と artisoc Cloud の違いを解説します。

戻り値のある関数

引数として `hikisuu1` と `hikisuu2`（整数型）をとり、実数型の戻り値を持つ関数 `kansuu` を作成する場合を考えます。

artisoc4

artisoc4 では以下のように `function` で関数を定義します。

```
function kansuu(hikisuu1 as integer, hikisuu2 as integer) as double {
    [処理]
    return [戻り値]
}
```

呼び出す場合には、同じルールエディタ内で単に関数名と引数を記述します。

```
a = kansuu(3, 5)  //関数を呼び出し、戻り値を変数 a に格納
```

artisoc Cloud

artisoc Cloud では、関数は以下のように `def` で定義します。１つ目の引数として必ず `self` を指定することに注意が必要です。引数や戻り値の型を指定する必要はなく、処理はインデント内に記述します。

```
# ルールエディタ内に記述
def kansuu(self, hikisuu1, hikisuu2):
    [処理]
    return [戻り値]
```

関数を呼び出すときは、同じルールエディタ内で以下のように記述します。`self.[関数名]`の形で記述することや、引数には定義時と違って `self` を記述する必要がないことに注意します。

```
a = self.kansuu(3, 5)  # 関数を呼び出し、戻り値を変数 a に格納
```

戻り値のない関数（サブルーチン）

引数として`hikisuu1`と`hikisuu2`（整数型）をとり、戻り値を持たない関数（サブルーチン）を作成する場合を考えます。

artisoc4

artisoc4 では以下のように`sub`で定義します。

```
sub kansuu(hikisuu1 as integer, hikisuu2 as integer) {
    [処理]
}
```

呼び出す場合には、同じルールエディタ内で単に関数名と引数を記述します。

```
kansuu(3, 5)  //関数の呼び出し
```

artisoc Cloud

artisoc Cloud では戻り値の有無にかかわらず`def`で関数を定義します。戻り値のある関数との違いは`return`の有無だけです。１つ目の引数として必ず`self`を指定することに注意が必要です。引数の型を指定する必要はなく、処理はインデント内に記述します。

```
def kansuu(self, hikisuu1, hikisuu2):
    [処理]
```

関数を呼び出すときは、同じルールエディタ内で以下のように記述します。`self.[関数名]`の形で記述することや、引数に`self`を記述する必要がないことに注意します。

```
self.kansuu(3, 5)  # 関数の呼び出し
```

ファイル入出力

ファイル入出力の方法について、artisoc4 と artisoc Cloud の違いを解説します。

ファイル読み込み

以下のようなユースケースを想定して説明します。

- 以下のような csv 形式のファイル `input.csv` の各行を順番に読み込み、2 番目の要素を取得してコンソールにプリントする

input.csv

```
1, 2, 3, 4, 5
6, 7, 8, 9, 10
11, 12, 13, 14, 15
```

ここでは説明の都合上単にプリントしていますが、実際にはファイルから取得した数値を変数に格納するようなケースが考えられます。

artisoc4 の場合

artisoc4 では、たとえば以下のように記述します。

```
Dim line as String
Dim data as String
Dim value as Integer

OpenFile("input.csv", 1, 1)
Do While(IsEofFile(1) == False)
    line = ReadFile(1)
    data = GetToken(line, 2)
```

```
    value = CInt(data)
    PrintLn(value)
Loop
CloseFile(1)
```

変数定義の後、処理の 1 行目で `OpenFile` 関数を用いてファイルを読み込みます。2 番目の引数はファイル識別のための番号で、何でもかまいませんが 1 を指定しています。3 番目の引数はオープンモードで、ここでは読み込みなので 1 を指定します。ちなみに書き込みでは 2、追記では 3 を指定します。

2 行目では `IsEofFile` 関数を用いてファイル終了を判定し、while 文を用いて最終行までの繰り返し処理を行っています。

3 行目では `ReadFile` 関数を用いて 1 行を読み込んでいます。たとえば最初の繰り返しでは、変数 line には `1, 2, 3, 4, 5` というように最初の行のデータが入ります。次の繰り返しではその次の行のデータが入ります。

4 行目では `GetToken` 関数を用いて line の 2 番目の要素を取得し変数 data に格納します。最初の繰り返しでは、変数 data には `3` が入ります（要素番号は 0 番目から数えることに注意します）。また、ここで変数 data は文字列型であることに注意します。

5 行目では、`CInt` 関数を用いて文字列型の変数 data を整数型に変換しています。（今回はコンソールに出力するだけのであまり意味がありませんが、数値データとして用いるならばこの処理が必要です）

6 行目でコンソールにプリントします。2 行目から 6 行目の処理を csv ファイルの各行に対して繰り返します。

最後に `CloseFile` 関数を用いてファイルを閉じます。

実行するとコンソールには以下のように出力されます。

```
3
8
13
```

artisoc Cloud の場合

まったく同じ処理を行う場合、artisoc Cloud ではたとえば以下のように記述します。

```
import csv
with open("input.csv", mode="r") as f:
    reader = csv.reader(f)
    for line in reader:
        data = line[2]
        value = int(data)
        print(data)
```

1 行目で `csv` モジュールをインポートしています。これは csv ファイルを扱うのに便利な機能が入ったもので、このあと使用します。

2 行目で `with open` を用いてファイルを読み込み、変数 `f` に格納しています。`mode="r"` でオープンモードを読み込みに設定しています。このあとの処理はインデント内に記述することに注意します。

3 行目で `csv` モジュールを用いて、`reader` オブジェクトを作成しています。これは csv ファイルの各行を収めたデータであると考えてください。

4 行目で `for` 文を用い、csv ファイルの各行を変数 line に収めて繰り返し処理を行います。

5 行目で line の 2 番目の要素を変数 data に格納しています。たとえば最初の繰り返しでは、line は input.csv の最初の行ですので`[0, 1, 2, 3, 4, 5]`というリストになります。よって、data には `2` が入ります。ここで data は文字列型であることに注意します。

6 行目で `int` 関数を用いて文字列型の変数データを整数型に変換しています。

7 行目でコンソールにプリントします。5~7 行目の処理を csv ファイルの各行に対して繰り返します。

ファイル書き込み

以下のようなユースケースを想定して説明します。

- 毎ステップ終了時に、`output.csv` の 1 列目にステップ数を、2 列目に Universe 変数 `hensuu` 値を追記する

たとえば、5 ステップ目まで実行した場合には `output.csv` は以下のようになります。

output.csv

```
"1","0.6824156034474742",
"2","0.48215025483320517",
"3","0.0464513240693708",
"4","0.8707409926140838",
"5","0.9415978124186635",
```

変数の値の変化を各ステップで記録するイメージです。（ここでは変数値は単なる乱数としています）

artisoc4 の場合

artisoc4 では、Univ_Step_End に以下のように記述します。

```
OpenFileCSV("output.csv", 1, 3)
WriteFilecsv(1, GetCountStep(), False)
WriteFileCSV(1, Universe.hensuu, True)
CloseFilecsv(1)
```

1 行目で `OpenFileCSV` 関数を用いてファイルを開きます。2 番目の引数はファイル識別のための番号で、何でもかまいませんが 1 を指定しています。3 番目の引数はオープンモードで、ここではファイルの末尾に追記していくため 3 を指定します。（ちなみに 1 は読み込み、2 はファイル内のデータを破棄してから書き込みます）

2 行目で `WriteFileCSV` 関数を用いてステップ数を書き込みます。3 番目の引数は改行するかどうかで、ここでは False を指定しているので、次に `WriteFileCSV` が呼ばれたときは同じ行に書き込まれます。

3 行目で変数の値を書き込みます。3 番目の引数に True を指定しているので、次に `WriteFileCSV` が呼ばれたときは次の行に書き込まれます。

4 行目でファイルを閉じます。Univ_Step_End に記述しているため、ここまでの処理が毎ステップ終了時に実行されます。

artisoc Cloud の場合

まったく同じ処理を行う場合、artisoc Cloud では univ_step_end にたとえば以下のように記述します。

```
import csv
with open("output.csv", mode="a") as f:
    writer = csv.writer(f)
    data = [count_step(), Universe.hensuu]
    writer.writerow(data)
```

1 行目で `csv` モジュールをインポートしています。これは csv ファイルを扱うのに便利な機能が入ったもので、このあと使用します。

2 行目で `with open` を用いてファイルを読み込み、変数 `f` に格納しています。`mode="a"` でオープンモードを追記に設定しています（ちなみに `mode="r"` で読み込み、`mode="w"` でファイル内のデータを破棄してからの書き込みになります）。このあとの処理はインデント内に記述することに注意します。

3 行目で `csv` モジュールを用いて、`writer` オブジェクトを作成しています。これは csv ファイルに書き込むための道具を作ったものと考えてください。

4 行目で書き込むためのデータをリスト形式で用意しています。0 番目の要素にステップ数を、1 番目の要素に変数を格納しています。

5 行目で `writer` オブジェクトの `writerow` 関数を用いて data をファイルに書き込みます。これで、1 列目にステップ数が、2 列目に変数が追記されます。