**目次**

**第一章 潜竜、用うる勿れ**

**1. 変数とデータ型**

**1.1 変数**

**1.2 演算子**

**1.3 基本のデータ型**

**1.3.1 ブール値(bool)**

**1.3.2 整数(int)**

**1.3.3 浮動小数点数(float)**

**1.3.4 文字列(str)**

**1.4 データ構造**

**1.4.1 リスト(list)**

**1.4.2 タプル(tuple)**

**1.4.3 辞書(dict)**

**1.4.4 集合(set)**

**2. 制御構造**

**2.1 if elif else**

**2.2 for**

**実践**

**第二章 見龍田に在り**

**3. 関数**

**3.1 組み込み関数の呼び出し**

**3.2 自分の関数を定義**

**3.3 引数(arguments)&パラメータ(parameters)**

**3.3.1 位置引数**

**3.3.2 キーワード引数**

**3.3.3 デフォルト引数**

**3.3.4 可変長引数**

**3.4 再帰関数**

**4. エラー処理**

**4.1 例外を投げる**

**4.2 try except else finally**

**5. 入出力**

**5.1 標準入出力**

**実践**

**第三章 終日乾乾す**

**6. オブジェクト指向プログラミング（Object Oriented Programming）**

**6.1 クラス(Class)とオブジェクト(object)**

**6.1.1関数型プログラミング(Functional Programming)**

**6.1.2オブジェクト指向プログラミング(Object Oriented Programming)**

**6.1.3 classによるクラスの定義**

**6.1.4 Pythonのコンストラクタ(Constructor): \_\_init\_\_メソッド**

**6.1.5 Pythonのデストラクタ(Destructor):\_\_del\_\_メソッド**

**6.1.6あるオブジェクトが特定のクラスのインスタンスかチェックしよう**

**6.2 アクセス制御**

**6.2.1アトリビュート(Attribute)の操作について**

**6.2.2アトリビュートの有無チェック**

**6.2.3インスタンスアトリビュート(Instance Attributes)と**

**クラスアトリビュート(Class Attributes)**

**6.2.4非公開アトリビュートのための名前修飾(Name Mangling)**

**6.2.5プロパティ(property)によるアトリビュートのコントロール**

**6.2.6 \_\_slots\_\_によるアトリビュート名のコントロール**

**6.3 継承（Inheritance）**

**6.3.1一つのクラスの継承**

**6.3.2複数クラス継承**

**6.4 ポリモーフィズム（Polymorphism）**

**実践**

**付録　豆知識の吹き寄せ**

1. **リストの内包表記(List Comprehensions)**
2. **オブジェクトを初期化するとき、はじめて実行されるメソッドは\_\_init\_\_()メソッドではない**
3. **標準的なクラスの定義方法について**

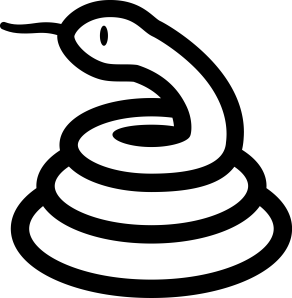
**参考文献**

**参考文献**

「Python」という英語の単語の意味はニシキヘビ（熱帯地方の巨大な蛇）です。ニシキヘビはPython言語とは何の関係もないが、しばしば誤解されます。但是在我看来，Python确实拥有巨蟒的某些特性。比如它看起来很迟缓，但是却异常迅猛。

随着网络技术的发展和机器学习算法开发的普及，Python已经成为了现在最热门的编程语言之一。

東洋文化の中で、巨大な蛇は常に龍の伝説に関連しています。

****

**第一章**

**潜竜、用うる勿れ**

1. **変数とデータ型**

# **変数**

プログラミングで使う変数とは、メモリ上のデータに、アクセスしやすいようにつける名前です。数学の変数も3x+1=4のxの部分を指しますが、基本的にプログラミングの変数とは違うものとして考えましょう。

x = 100

a = b = 233

このように、変数を左、代入する値を右に書きます。

イコールの両端の半角スペースは、開けても開けなくてもかまいません。

c=12

d =12

e= 12

一つのイコールでも、カンマで区切れば複数の変数に複数の値を一度に代入できます。

brother, sister = 19, 18

print(brother, sister)

>>> 19 18

ちなみに、変数名として使えるのは

* 小文字の英字
* 大文字の英字
* 数字
* アンダースコア

のみで、数字は変数の名前の先頭には使えません。（**※注意：大文字、小文字は区別する**）

My\_brother = 19

\_your\_ = 22

My\_brother2 = 20

\_12m = '12 month'

これらは有効です。

**※　注意：数字は変数の名前の先頭には使えません。**

3a = 100 # 無効

3\_a = 10 # 無効

変数の名前はルールに則れば、基本的には自由につけることができます。しかし、予約語や、組み込み関数と同じ名前をつけるのは避けましょう。Pythonには、「def、class、True、False、import」など、30個以上の予約語があります。不用意に使うことでコード内で衝突が起こり、不具合を起こす可能性も無いとは言い切れません。

**Pythonの変数には関数も入れることができます。**

def foo():

print('Hello World')

x = foo

foo()

x()

>>> Hello World

Hello World

これは関数がオブジェクトとして扱えるからです。またクラスもオブジェクトなので、関数同様に変数に代入することができます。

Pythonでは変数の型を宣言する必要はありません。動的型付けによって、型が自動的に判別されています。変数に代入されたオブジェクトの型をしっかり確認したい場合は、typeとisinstance関数を使います。

x = 100

type(x)

>>> int

y = 'Hello World'

type(y)

>>> str

**※　宣言をしなければならない言語と区別してください。**

これらは無効です。

int x = 100　　# 無効

char y = 'H'　　# 無効

# **演算子**

演算は、平たく言うと、四則演算、比較演算、論理演算などの計算機能のことです。プログラミングでは、演算子という記号を用いて、この計算処理を行います。

**数値演算子一覧**

| **演算子** | **例** | **説明** |
| --- | --- | --- |
| + | a + b | 足し算 |
| – | a – b | 引き算 |
| \* | a\*b | 掛け算 |
| / | a/b | 割り算 |
| // | a//b | aをbで割った商の整数値 |
| % | a%b | aをbで割った時の割り切れなかった余り |
| \*\* | a\*\*n | aをn回掛けた数（べき乗） |

実際に見ていきましょう。

a = 10

b = 11

a + b

>>> 21

a - b

>>> -1

a \* b

>>> 110

a / b

>>> 0.9090909090909091

a//b

>>> 0

a % b

>>> 10

a \*\* b

>>> 100000000000

**文字列演算子**

| **演算子** | **例** | **説明** |
| --- | --- | --- |
| + | "a" + "b" | 文字列”a”と”b”を連結して、文字列”ab”になる |
| \* | "abc"\*n | 文字列”abc”をn回繰り返す |

実際に見ていきましょう。

First\_Name = "山田"

Last\_Name = "太郎"

First\_Name+ Last\_Name

>>> '山田太郎'

First\_Name\*3

>>> '山田山田山田'

**比較演算子の一覧**

| **演算子** | **例** | **説明** |
| --- | --- | --- |
| == | a == b は偽 | 2つの値がイコールの時に「真」 |
| != | a != b は真 | 2つの値がイコールでない時に「真」 |
| > | a > b は偽 | 左の値が右の値より大きい時に「真」 |
| < | a < b は真 | 左の値が右の値より小さい時に「真」 |
| >= | a >= b は偽 | 左の値が右の値以上の時に「真」 |
| <= | a <= b は真 | 左の値が右の値以下の時に「真」 |
| is | a is b は偽 | 2つのオブジェクトが同一の時に「真」 |
| is not | a is not b は真 | 2つのオブジェクトが同一でない時に「真」 |

実際に見ていきましょう。

a = 1

b = 100

if a == b:

print('Yes')

else:

print('No')

>>> No

if a != b:

print('Yes')

else:

print('No')

>>> Yes

if a < b:

print('Yes')

else:

print('No')

>>> Yes

if a > b:

print('Yes')

else:

print('No')

>>> No

if a <= b:

print('Yes')

else:

print('No')

>>> Yes

if a >= b:

print('Yes')

else:

print('No')

>>> No

if a is b:

print('Yes')

else:

print('No')

>>> No

if a is not b:

print('Yes')

else:

print('No')

>>> Yes

**論理演算子の一覧**

| **演算子** | **例** | **説明** |
| --- | --- | --- |
| and | a and b | 論理積。aとbの両方が「真」の時「真」。一方でも「偽」なら「偽」。 |
| or | a or b | 論理和。aかbのどちらか一方でも「真」の時「真」。両方とも「偽」なら「偽」。 |
| not | not a | 否定。　aが「真」なら「偽」。aが「偽」なら「真」。 |

実際に見ていきましょう。

a = 1

b = 3

print(a > 2 and b < 3)

>>> False

print(a > 2 or b < 3)

>>> False

print(not b < 3)

>>> True

# **基本のデータ型**

### **ブール値(bool)**

a = True

type(a)

>>> bool

ｂ= False

type(b)

>>> bool

**TrueとFalseはブール値と呼ばれます。**

**ブール型と数値との関係**

Pythonのブール型は、Falseは' '、""、0、()、[]、{}、None。

Trueはこれら以外の任意の値として振舞うことができます。

if 0:

print("True")

else:

print("False")

>>> False

if 23:

print("True")

else:

print("False")

>>> True

if '0':

print("True")

else:

print("False")

>>> True

if None:

print('True')

else:

print("False")

>>> False

if 'False':

print('True')

else:

print("False")

>>> True

### **整数(int)**

整数型とは、一般的にはint型と呼ばれているものです。

int型は整数をもつオブジェクトのことです。

b = 1

type(b)

>>> int

### **浮動小数点数(float)**

浮動小数点数型とは、一般的にはfloat型と呼ばれているものです。

float型は小数をもつオブジェクトのことです。

c = 3.1415926

type(c)

>>> float

### **文字列(str)**

文字列型は文字列を値としてもっているオブジェクトのタイプのことです。

文字列とは、ダブルクォート" かシングルクォート' で文字を囲んだものです。

d = "I Love Python"

type(d)

>>> str

**※ やさしい文字列の操作方法**

**① 次の構文を使用すると、開始文字・終了文字を調べることができます。**

filename = 'test.txt'

filename.endswith('.txt')

>>> True

url = 'http://www.python.org'

url.startswith('http')

>>> True

startswithは文字列の先頭を調べるのに対し、endswithは文字列の末尾を調べます。 検索する文字列が存在していれば、Trueになり、そうでなければ、Falseとなります。

**② 要素のインデックス(何番目か)**

文字列のなかの一つの文字を取り出したいときには、文字列名の後ろに角かっこ[ ]で囲んだ文字のオフセットを続けます。先頭(もっとも左)の文字のオフセットは0、その右が1、というように数えます。末尾(もっとも右)の文字のオフセットは-1でも指定できるので、文字数を数える必要はありません。右端の左は-2、さらにその左は-3のように続きます。

d = "I Love Python"

d[0]

>>> 'I'

d[1]

>>> ' '

d[2]

>>> 'L'

d[3]

>>> 'o'

d[4]

>>> 'v'

d[-1]

>>> 'n'

d[-2]

>>> 'o'

d[-3]

>>> 'h'

**※ 文字列の長さ以上のオフセットを指定すると、エラーが出ます。(オフセットは0から長さ-1までの範囲になることをわすれないでください)**

**③ [start:end:step]によるスライス**

スライスを使えば、文字列から部分文字列(文字列の一部)を取り出すことができます。スライスは、角かっこと先頭オフセット(start)、末尾オフセット(end)、ステップ(step)で定義します。これらのなかには省略できるものがあります。スライスには、startからendの１字手前までの文字が含まれます。

* [:]は、先頭から末尾までのシーケンス全体を抽出する。
* [start:]は、startオフセットから末尾までのシーケンスを抽出する。
* [:end]は、先頭からend - 1オフセットまでのシーケンスを抽出する。
* [start:end]は、startオフセットからend - 1オフセットまでのシーケンスを抽出する。
* [start:end:step]は、startオフセットからend - 1オフセットまでのシーケンスをstep文字おきに抽出する。

**※　基本なシーケンス(sequence):　文字列(str)、リスト(list)、タプル(tuple)、range**

d = "I Love Python"

d[:]

>>> 'I Love Python'

d[1:]

>>> ' Love Python'

d[-4:]

>>> 'thon'

d[:6]

>>> 'I Love'

d[2:6]

>>> 'Love'

d[2:10:3] # 2から9までの部分文字列を3文字おきに抽出する

>>> 'Ley'

**④ split()による分割**

数字、アルファベットや記号などが入り混じった文字列を、ある規則に従って、切り分けてリスト化してくれるのが、split()関数です。文字列名.split('区切り文字') とすることで、区切り文字で区切られたリストを得る事が出来ます。

d = "I Love Python"

d.split(' ') # ' 'で区切られたリストを得る

>>> ['I', 'Love', 'Python']

**⑤ 大文字と小文字**

d = "I Love Python"

d.upper() # すべての文字を大文字に変換する

>>> 'I LOVE PYTHON'

d.lower() # すべての文字を小文字に変換する

>>> 'i love python'

**多彩な文字列操作**

Pythonは、ここに示したものよりはるかに多くの文字列関数を持っています。文字列の操作はプログラミングの中に常に使われる技術の一つです。すべての文字列関数は以下通りです。

[x for x in dir(str) if not x.startswith('\_\_')]

>>> ['capitalize',

'casefold',

'center',

'count',

'encode',

'endswith',

'expandtabs',

'find',

'format',

'format\_map',

'index',

'isalnum',

'isalpha',

'isdecimal',

'isdigit',

'isidentifier',

'islower',

'isnumeric',

'isprintable',

'isspace',

'istitle',

'isupper',

'join',

'ljust',

'lower',

'lstrip',

'maketrans',

'partition',

'replace',

'rfind',

'rindex',

'rjust',

'rpartition',

'rsplit',

'rstrip',

'split',

'splitlines',

'startswith',

'strip',

'swapcase',

'title',

'translate',

'upper',

'zfill']

# **データ構造**

### **リスト(list)**

リストとは、複数の要素を含むコンテナ型オブジェクト（容器）の事を指します。

**リストの作成と初期化**

lt = []

# または

lt = list()

また、初期化と同時に要素を追加することも出来ます。

lt1 = [1,2,3,'name'] # listのなかに何でも追加できます。

lt2 = [1,2,'name',[23,21],{'name':'Xu hongkun'}]

lt3 = list('I Love Python')

lt1[3]

>>> 'name'

lt2[3][0]

>>> 23

lt2[4]['name']

>>> 'Xu hongkun'

lt3

>>> ['I', ' ', 'L', 'o', 'v', 'e', ' ', 'P', 'y', 't', 'h', 'o', 'n']

**リストの操作方法**

[a for a in dir(list) if not a.startswith('\_\_')] # すべてのリストの操作方法一覧

>>> ['append',

'clear',

'copy',

'count',

'extend',

'index',

'insert',

'pop',

'remove',

'reverse',

'sort']

1. append()による末尾への要素の追加

lt1 = [1,2,3,'name']

lt1.append('NewOne')

lt1

>>> [1, 2, 3, 'name', 'NewOne']

lt1.append(['One', 'Two'])

lt1

>>> [1, 2, 3, 'name', 'NewOne', ['One', 'Two']]

1. extend()または+=を使ったリストの結合

others = ['1', '2', '3']

lt1.extend(others)

lt1

>>> [1, 2, 3, 'name', 'NewOne', ['One', 'Two'], '1', '2', '3']

lt1 += others

lt1

>>> [1, 2, 3, 'name', 'NewOne', ['One', 'Two'], '1', '2', '3', '1', '2', '3']

1. index()により要素の値から要素のオフセットを知る方法

ls = ['okida', 'akira', 'suda', 'tachibana']

ls.index('akira')

>>> 1

1. insert()によるオフセット(offset)を指定した要素の追加

append()関数は、リストの末尾にしか要素を追加できない。リストのオフセットを指定し、その前に要素を追加したいときには、insert()を使う。

ls = ['okida', 'akira', 'suda', 'tachibana']

ls.insert(1, 'suenaga')

ls

>>> ['okida', 'suenaga', 'akira', 'suda', 'tachibana']

1. remove()による値に基づく要素の削減

ls.remove('suda')

ls

>>> ['okida','suenaga', 'akira', 'tachibana']

ls.remove('suda') # もう一度'suda'を削除してみてください。

>>> ---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-60-f155de4a3534> in <module>

----> 1 res = ls.remove('suda')

2 print()

ValueError: list.remove(x): x not in list

help(list.remove)

>>> Help on method\_descriptor:

remove(...)

L.remove(value) -> None -- remove first occurrence of value.

Raises ValueError if the value is not present.

try:

ls.remove('suda')

except ValueError as e:

print('この値はない')

finally:

pass

>>> この値はない

1. pop()でオフセットを指定して要素を取り出し、削除する方法

ls = ['okida', 'akira', 'suda', 'tachibana']

ls.pop()

>>> 'tachibana'

ls

>>> ['okida', 'akira', 'suda']

ls.pop(1)

>>> 'akira'

ls.pop(-1)

>>> 'suda'

ls

>>> ['okida']

1. count()を使った値の個数の計算

ls = ['okida', 'akira', 'akira', 'tachibana']

ls.count('akira')

>>> 2

ls.count('hijiri')

>>> 0

1. copy()によるコピー

ls1 = ['okida', 'akira', 'tachibana']

ls = ls1 # リストにとって代入はコピーではない、元の値が変わる

ls[0] = 123456

ls1

>>> [123456, 'akira', 'tachibana']

ls1 = ['okida', 'akira', 'tachibana']

ls2 = ls1.copy()

ls2[0] = 'watasi'

ls1

>>> ['okida', 'akira', 'tachibana']

ls2

>>> ['watasi', 'akira', 'tachibana']

**豆知識：リストの内包表記(List Comprehensions)**

リストの内包表記(List Comprehensions)を使うとリスト(list)の作成が簡単にかけます。主な利用場面は、あるシーケンスのそれぞれの要素に対してある操作を行った結果を要素にしたリストを作ったり、ある条件を満たす要素だけからなる部分シーケンスを作成することです。例えば、次のような平方のリストを作りたいとします:

# 普通の方法

temp = [1, 4, 2, 3]

result = []

for x in temp:

result.append(x\*\*2)

print(result)

>>> [1, 16, 4, 9]

# リストの内包表記

temp = [1, 4, 2, 3]

result = [x\*\*2 for x in temp]

print(result)

>>> [1, 16, 4, 9]

これまで、文字列の操作方法一覧とリストの操作方法一覧はいずれもこの方法で生成されるものです。

### **タプル(tuple)**

タプルは、リストと同様に任意の要素を集めたシーケンスです。リストとは異なり、タプルは**イミュータブル(Immutable)**です。つまり、タプルを定義したあとで要素を追加・削除・変更できません。そのためタプルは、定数リストとも言われます。

**タプルの作成と初期化**

tp1 = ()

type(tp)

>>>　tuple

# または

tp2 = (1, 2, 3, 'her') # tupleのなかに何でも追加できます。

tp2

>>> (1, 2, 3, 'her')

# または

tp3 = 1, 2, 3, 'her'

tp3

>>> (1, 2, 3, 'her')

type(tp3)

>>> tuple

# 要素は1個のタプルでも末尾にカンマを付けて作る

tp4 = 3,

tp4

>>> (3,)

type(tp4)

>>> tuple

**タプルの操作方法**

[x for x in dir(tuple) if not x.startswith('\_\_')] # タプルの操作方法一覧

>>> ['count', 'index']

### **辞書(dict)型**

辞書（dictionary）型とは、{ }の中にkeyとvalueの組み合わせが含まれているデータのことです。

{'key1':'value1', 'key2':'value2', 'key3':'value3'}

のように、辞書型のオブジェクトを構成する要素はkeyとvalueをコロンで区切ったペアになっています。

※ valueは何の型の値でもいい。一方で、keyは変更が出来ない型に限定されます。

new = {1:23, 2:32, (23,12):23}

print(new)

print(new[(23, 12)])

>>> {1: 23, 2: 32, (23, 12): 23}

23

**辞書型オブジェクトの作り方**

empty\_dict = {}

type(empty\_dict)

>>> dict

mydict = {'apple':1, 'banana':2, 'orange':3}

mydict

>>> {'apple': 1, 'banana': 2, 'orange': 3}

# なぜ辞書と呼びますか？この形で書くと、辞書に似ているでしょう。

dictionary = {

'class':'People having the same social or economic status',

'inheritance':'Any attribute or immaterial possession that is inherited from ancestors.',

'polymorphism':'The existence of two or more forms of individuals within the same animal species',

}

dictionary

>>> {'class': 'People having the same social or economic status',

'inheritance': 'Any attribute or immaterial possession that is inherited from ancestors.',

'polymorphism': 'The existence of two or more forms of individuals within the same animal species'}

# dict()を使った変換

l1 = [['a', 1], ['b', 2], ['c', 3]] # 2要素のリストのリスト

l2 = [('a', 1), ('b', 2), ('c', 3)] # 2要素のタプルのリスト

l3 = (['a', 1], ['b', 2], ['c', 3]) # 2要素のリストのタプル

l4 = ['a1', 'b2', 'c3'] # 2字の文字列のリスト

l5 = ('a1', 'b2', 'c3') # 2字の文字列のタプル

mydict1 = dict(l1)

print('mydict1 :', mydict1) # mydict1

mydict2 = dict(l2)

print('mydict2 :', mydict2) # mydict2

mydict3 = dict(l3)

print('mydict3 :', mydict3) # mydict3

mydict4 = dict(l4)

print('mydict4 :', mydict4) # 注意: mydict4のキー

mydict5 = dict(l5)

print('mydict5 :', mydict5) # 注意: mydict5のキー

>>> mydict1 : {'a': 1, 'b': 2, 'c': 3}

mydict2 : {'a': 1, 'b': 2, 'c': 3}

mydict3 : {'a': 1, 'b': 2, 'c': 3}

mydict4 : {'a': '1', 'b': '2', 'c': '3'}

mydict5 : {'a': '1', 'b': '2', 'c': '3'}

**辞書型の要素にアクセスする方法**

辞書型オブジェクトは、リスト型のように多くの要素を持つことが出来ます。しかしリスト型とは違って辞書型オブジェクトには順序がありません。なので、要素を取り出すためにはkeyを使うことになります。

dictionary = {

'class':'People having the same social or economic status',

'inheritance':'Any attribute or immaterial possession that is inherited from ancestors.',

'polymorphism':'The existence of two or more forms of individuals within the same animal species',

}

dictionary['inheritance']

>>> 'Any attribute or immaterial possession that is inherited from ancestors.'

mydict = {'apple':1, 'banana':2, 'orange':3}

mydict['apple']

>>> 1

**Pythonの辞書型オブジェクトでは、存在しないkeyを指定するとエラーが発生するので気をつけましょう。**

mydict = {'apple':1, 'banana':2, 'orange':3}

mydict['peach']

>>> ---------------------------------------------------------------------------

KeyError Traceback (most recent call last)

<ipython-input-13-d0fdd0540e4b> in <module>

1 mydict = {'apple':1, 'banana':2, 'orange':3}

----> 2 mydict['peach']

KeyError: 'peach'

**辞書型に要素を追加する方法**

[x for x in dir(dict) if not x.startswith('\_\_')] # すべての辞書操作方法一覧

>>> ['clear',

'copy',

'fromkeys',

'get',

'items',

'keys',

'pop',

'popitem',

'setdefault',

'update',

'values']

### **集合(set)型**

Pythonのset型とは、集合を扱うための型です。もっとも普及しているリスト型のように、set型も同じく複数の値を格納できる型です

しかし、リスト型との決定的な違いは

* **重複した要素がない**
* **要素に順番がない**

といった二点が挙げられます。

**集合型オブジェクトの作り方**

myset = set()

myset

>>> set()

myset1 = {1, 'set', "This is a apple", 1, 1, 1, 1, 1}

myset1 # 重複した要素がない

>>> {1, 'This is a apple', 'set'}

mydict = { } # これはdictです、setではありません

type(mydict)

>>> dict

# リストから集合へ

myset\_a = set([1, 'set', 234, 'set'])

# タプルから集合へ

myset\_b = set((1, 'set', 234))

# set()に辞書を渡すと、キーだけが使われます

myset\_c = set({"apple":1, "orange":2, "banana":3})

print(myset\_a)

print(myset\_b)

print(myset\_c)

>>> {1, 234, 'set'}

{1, 234, 'set'}

{'banana', 'orange', 'apple'}

**集合の操作方法**

[x for x in dir(set) if not x.startswith('\_\_')] #　集合の操作方法の一覧

>>> ['add',

'clear',

'copy',

'difference',

'difference\_update',

'discard',

'intersection',

'intersection\_update',

'isdisjoint',

'issubset',

'issuperset',

'pop',

'remove',

'symmetric\_difference',

'symmetric\_difference\_update',

'union',

'update']

1. **制御構造**

# **if elif else**

1. **if文の使い方**

if文はプログラムの中で条件分岐を書きたいときに使う文です。

if文は、与えられた条件が、

* Trueであるか？
* Falseであるか

を確認します。if文を使った条件分岐は、プログラミングそのものの基本的な考え方の一つになります。

if文の基本的な構文はこのようになります。

**if 条件式:**

**条件式がTrueのときに行う処理**

この条件式がTrueのときにif文のブロック（インデントが下がっている部分）に書かれた処理を実行します。インデントとは、行頭に空白を入れて文字を入れて字下げを行う事を言います。Pythonは同じ数の空白でインデントされたまとまりを一つのブロックと認識します。1レベルインデントするごとに、スペース(半角)を4つ使いましょう。

例：

x = 100

if x > 10:

print("It's True")

>>> It's True

条件式は論理演算子や比較演算子で計算されて、その式はTrueかFalseを返します。

※ **Pythonのfor文には{と}がありません。そして、インデントが必要です**

x = 1

if x > 10:

print("It's True")

print("I don't know!") # 4つのスペース(半角)がないと、一つのブロックと認識されていません。

>>> I don't know!

1. **if elseの使い方**

先程のサンプルコードは、条件式がTrueになるときのみの処理を用意しました。今度はTrueのときだけでなく、条件式がFalseになったときに動く処理を追加しましょう。条件を満たさなかったときの処理を書くには、elseを使います。

elseを使った基本的な構文はこのようになります。

**if 条件式:**

**条件式がTrueのときに行う処理**

**else:**

**条件式がFalseのときに行う処理**

* **elseは条件式を持ちません。**

例：

x = 100

if x > 10:

print('x > 10')

else:

print('x <= 10')

>>> x > 10

1. **if elif else文の使い方**

「if文の条件には引っかからなかったけど、別の条件に当てはまったらelseとは別の処理がしたい」といったときはelifで実現できます。

**if 条件式1:**

**条件式がTrueのときに行う処理**

**elif 条件式2:**

**elifの条件式2がTrueのときに行う処理**

**else:**

**if文の条件式1もelifの条件式2もどちらもFalseのときに行う処理**

使い方はif文のときとほぼ変わらず、条件式を持つことができます。

x = 100

if x > 10 and x < 100:

print('10 < x < 100')

elif x >= 100:

print('x >= 100')

else:

print('x <= 10')

>>> x >= 100

# **for**

for文を使えば、例えば、1から100までの数字を順番に出力したり、配列の要素を順番に取り出したりすることができます。

**for文の書き方**

**for 変数 in オブジェクト:**

**オブジェクトの中にこの変数があるときの処理**

実際に見ていきましょう。

fruits = ['apple', 'banana', 'orange']

for x in fruits:

print('I have a ', x)

>>> I have a apple

I have a banana

I have a orange

*プロセス：*

*変数xに、配列fruitsの最初の要素（apple）が代入されてprintで出力*

*変数xに、配列fruitsの二番目の要素（banana）が代入されてprintで出力*

*変数xに、配列fruitsの三番目の要素（orange）が代入されてprintで出力*

**range()関数の使い方**

**先ほどは、fruitsという配列を使って、繰り返す回数を指定していました。しかし、for文では、そのような繰り返し方だけではなく、単に10回繰り返す、100回繰り返すなどの繰り返し方もできます。そのような場合には、range**()**関数という関数を使うと便利です。**

**例えば、10回繰り返すfor文を書きたい場合は、以下のようにコードを記述します。**

for x in range(10):

print(x, ' I Love You!')

>>> 0 I Love You!

1 I Love You!

2 I Love You!

3 I Love You!

4 I Love You!

5 I Love You!

6 I Love You!

7 I Love You!

8 I Love You!

9 I Love You!

range()関数は何から何までの範囲を指定することもできます。

for x in range(5, 13):

print(x)

>>> 5

6

7

8

9

10

11

12

さらに、range()関数は、増加量もコントロールできます。

for x in range(0, 13, 3):

print(x)

>>> 0

3

6

9

12

range(0, 13, 3)の3は、増加量を表しています。つまり、0以上13未満の数字の中で、3ずつ増加するということなので、「0, ３, ６, ９, 12」が出力されているのです。

**breakの使い方**

for文を使っていると、繰り返し処理を途中でやめたいという場合、breakというものを使います。

for x in range(10):

print(x)

if x == 5:

print('x == 5')

break

print('next >>')

>>> 0

next >>

1

next >>

2

next >>

3

next >>

4

next >>

5

x == 5

**continueの使い方**

continue文は、continueの後にあるコードの部分をスキップする形でfor文を次に進めることができます。

for x in range(10):

print(x)

if x == 5:

print('x == 5')

continue

print('next >>')

>>> 0

next >>

1

next >>

2

next >>

3

next >>

4

next >>

5

x == 5

6

next >>

7

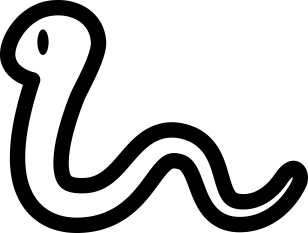
next >>

8

next >>

9

next >>

****

**第二章**

**見龍田に在り**

1. **関数**

関数とは、処理をまとめたものです。同様の処理を抜き出して共通化させ効率よくコーディングするため、処理を理解しやすくするために利用します。

# **組み込み関数の呼び出し**

Python インタプリタには数多くの関数と型が組み込まれており、いつでも利用できます。

Pythonの組み込み関数をここにアルファベット順に挙げます。

| **組み込み関数の一覧** | | | | |
| --- | --- | --- | --- | --- |
| abs() | dict() | help() | min() | setattr() |
| all() | dir() | hex() | next() | slice() |
| any() | divmod() | id() | object() | sorted() |
| ascii() | enumerate() | input() | oct() | staticmethod() |
| bin() | eval() | int() | open() | str() |
| bool() | exec() | isinstance() | ord() | sum() |
| bytearray() | filter() | issubclass() | pow() | super() |
| bytes() | float() | iter() | print() | tuple() |
| callable() | format() | len() | property() | type() |
| chr() | frozenset() | list() | range() | vars() |
| classmethod() | getattr() | locals() | repr() | zip() |
| compile() | globals() | map() | reversed() | \_\_import\_\_() |
| complex() | hasattr() | max() | round() |  |
| delattr() | hash() | memoryview() | set() |  |

これまでは、この中の多くの関数を使用してきました。まだ見たことのない関数もありますが、Pythonのhelp()関数をよく利用して、新しい組み込み関数を使ってみよう。

help(divmod)

>>> Help on built-in function divmod in module builtins:

divmod(x, y, /)

Return the tuple (x//y, x%y). Invariant: div\*y + mod == x.

divmod(3, 13)

>>> (0, 3)

すなわち、このタプルの意味は3 // 13 == 0と3 % 13 == 3ですね。

このように、help()を利用して関数を理解するのはいい方法だと思います。万が一、英語の説明が理解できないとき、また日本語のPythonの組み込み関数ドキュメントを見てください。でも、ほとんどの新しいPythonライブラリの関数はまだ日本語バージョンのドキュメントがないので、help()関数を忘れないでください。

# **自分の関数を定義**

Pythonで自分の関数(function)の定義方法は、def 関数名():というように定義する。関数内の処理は、インデントをずらして記述します。

# 引数なし

def add1():

pass # 何もしない

# 引数あり

def add2(a, b):

pass # 何もしない

※ **Pythonの関数には{と}がありません。そして、インデントが必要です**

def foo():

pass # インデントが必要です

foo()

>>> File "<ipython-input-22-9b6980ffa810>", line 2

pass

^

IndentationError: expected an indented block

Indentationエラーが起きてしまっています。

前の章に説明したようにインデントとは、行頭に空白を入れて文字を入れて字下げを行う事を言います。Pythonは同じ数の空白でインデントされたまとまりを一つのブロックと認識します。1レベルインデントするごとに、スペース(半角)を4つ使いましょう。

def foo(a, b): # 2つの変数の値を交換する

a, b = b, a

print(a, b)

このように、pythonでは括弧を使わずにインデントで処理をグループ化してまとめています。

**戻り値**

戻り値を定義しない場合、関数を実行しても返却されるのは、Noneです。

戻り値を定義したい場合、関数の最後にreturn 戻り値と記述します。

# 引数なし

def add3():

pass # 戻り値なし

# 引数あり

def add4(a, b):

return a, b # Pythonでは戻り値の数にかかわらず、戻り値をタプル(tuple)として返されます。

result1 = add3()

result2 = add4(1, 2)

print(result1)

print(result2)

print(type(result1))

print(type(result2))

>>> None

(1, 2)

<class 'NoneType'>

<class 'tuple'>

# **引数(arguments)&パラメータ(parameters)**

**パラメータ(parameters)とは**

パラメータ(parameters)とは関数に渡される具体的な値のプレースホルダ(Placeholder)です。パラメータ名を使うことで関数内で変数として参照できます。

def foo(arg1, arg2, arg3='default'): # arg1, arg2, arg3はパラメータ(parameters)です。

if arg3 == 'default':

return arg1 + arg2

else:

return arg1 - arg2

**引数(arguments)とは**

引数(arguments)とは関数に渡される具体的な値です。関数内での実際の計算に使用されます。

foo(3, 4) # 3, 4は引数です。

>>> 7

引数とパラメータはそれぞれ実引数、仮引数と呼ばれることもあります。

たまたま、引数はパラメータの意味でも使われることもありますが、ここでは、引数を値のプレースホルダという狭い意味で定義しておきます。

この記事のあとに、都合のために、"パラメータ"は関数を定義する時に関数に渡される具体的な値のプレースホルダ(Placeholder)を意味します。"引数"は関数に渡される具体的な値を意味します。

### **位置パラメータ**

Pythonは、関数に渡されたパラメータの処理ということでは、一般にほかの言語の多くと比べて柔軟性が高い。もっともよく知られたパラメータのタイプは、先頭から順に対応する位置の引数にコピーされる**位置パラメータ**です。

def food(first, second, third):

return {1:first, 2:second, 3:third}

food()関数は、位置パラメータから辞書を作って返します。

food('rice', 'noodle', 'soup')

>>> {1: 'rice', 2: 'noodle', 3: 'soup'}

food('noodle', 'soup', 'rice')

>>> {1: 'noodle', 2: 'soup', 3: 'rice'}

引数の位置によって、返した結果も違います。これは位置パラメータの特徴です。個々の位置のパラメータの意味を覚えておかなければならないという欠点があります。

### **キーワード引数**

位置パラメータの混乱を避けるには、対応する引数の名前を指定してパラメータをすればいいです。関数定義とパラメータの順序が異なっても構いません。

food(second='noodle', third='soup', first='rice')

>>> {1: 'rice', 2: 'noodle', 3: 'soup'}

**位置パラメータとキーワード引数は併用ができます。**

位置パラメータとキーワード引数の両方を使って関数を呼び出す場合には、まず先に位置パラメータを指定しなければなりません。

food('rice', third='soup', second='noodle')

>>> {1: 'rice', 2: 'noodle', 3: 'soup'}

### **デフォルトパラメータ**

Pythonのパラメータには、デフォルト値を設定することができます。設定することによって、関数はデフォルト値を設定したパラメータを指定しなくても実行することができます。その場合に、デフォルト値が有効になります。

def speak(human, content, language='Japanese'):

result = str(human) + ' speaks '+ str(content) + ' in ' + str(language) + '.'

print(result)

# 位置パラメータでパラメータを指定すれば、それがデフォルト値の代わりに使われます。

speak('Mrs.tatibana', 'Hello', 'English')

# キーワード引数でパラメータを指定すれば、それがデフォルト値の代わりに使われます。

speak('Mrs.yamazaki', 'Hello', language ='Chinese')

# デフォルト値を使います。

speak('Mrs.arimura', 'Hello')

>>> Mrs.tatibana speaks Hello in English.

Mrs.yamazaki speaks Hello in Chinese.

Mrs.arimura speaks Hello in Japanese.

**※　デフォルトパラメータのデフォルト値はリスト型と辞書型のデータを避けよう**

def foo(L=[]):

L.append('end')

return L

foo()関数は、指定されたリストに'end'要素を追加したい。指定されない場合には、新しいリストを作って、'end'要素を追加して、それを返却したい。

a = foo(["山田", "鈴木", "立花"]) # 位置パラメータでLが指定されました。何度も実行しても問題なし。

print(a)

>>> ['山田', '鈴木', '立花', 'end']

b = foo(L=["山田", "鈴木", "立花"]) # キーワード引数でLが指定されました。何度も実行しても問題なし。

print(b)

>>> ['山田', '鈴木', '立花', 'end']

c = foo() # Lが指定されていない。このような形で、foo()関数がはじめて呼び出されるのは問題ない。

print(c)

>>> ['end']

d = foo() # cと同じように、Lが指定されていない。['end']をほしいが...

print(d)

>>> ['end', 'end']

リストLを指定していないで、２回目に呼び出されたとき、結果には前回の呼び出しでセットされた１個の要素がまだ残っています。

デフォルト引数の値は関数定義時に一度だけ呼ばれ生成され、関数呼出し時には既に生成された値を使いまわしています。上のように、リストなどの値が変更できるオブジェクト(リスト型と辞書型のデータ)をデフォルト引数に与えており、引数指定をしなければ、すべての関数呼び出しでこのデータが共有されてしまいます。

**上記のfoo()関数の正しい書き方:**

def foo(L=None):

if L is None:

L = []

L.append('end')

return L

c = foo()

d = foo()

print(c)

print(d)

>>> ['end']

['end']

### **可変長引数**

可変長引数とは、「任意の数の引数を受けてるための引数」です。

**\*args:複数の引数をタプルとして受け取る**

Pythonで可変長引数を受け取りたい場合は、パラメータ名の前に\*を付けて関数を定義します。受け取った結果は、タプルとして関数内で使用できます。慣習では、可変長引数の名前を「args」としています。ただ、あくまでも慣習なのでargsという名称でなくとも動作します。

def add(\*arg):

print(arg)

sum = 0

for item in arg:

sum = sum + item

return sum

res1 = add(1,2,3,4,5) # どんなに多くの位置引数があっても問題がないです。

print(res1)

>>> (1, 2, 3, 4, 5)

15

res2 = add(1,2,3)

print(res2)

>>> (1, 2, 3)

6

res3 = add()

print(res3)

>>> ()

0

**\*\*kwargs: 複数のキーワード引数を辞書として受け取る**

もう一つの方法は、パラメータ名の前に\*\*を付けて関数を定義します。受け取った結果は、辞書として関数内で使用できます。「\*args」と同じ、慣習では、この時の可変長引数の名前を「kwargs」としています。ただ、あくまでも慣習なのでkwargsという名称でなくとも動作します。

def show(\*\*kwargs):

print(kwargs)

# items()が辞書型の組み込み関数(方法)の一つ。　詳しくはhelp(dict.items)

for item in kwargs.items():

print('item:', item)

# どんなに多くのキーワード引数があっても問題がないです。

show(first='Monkey', second='Elephant', third='giraffe' )

>>> {'first': 'Monkey', 'second': 'Elephant', 'third': 'giraffe'}

item: ('first', 'Monkey')

item: ('second', 'Elephant')

item: ('third', 'giraffe')

**また、可変長引数と通常の引数と併用も可能です。**

例：

def Max(Name, \*arg, \*\*kwargs):

for item in arg:

sum = 0

# values()が辞書型の組み込み関数(方法)の一つ。　辞書型データの値のリストを返却します。

for grade in kwargs.values():

sum = sum + grade # 合計得点

print(item + 'へ ')

print(Name + 'の成績は：')

print(kwargs)

print('合計:', str(sum))

print()

Max('山田', '山田さんのお父さん', '山田さんのお母さん', Math=100, English=60, Japanese=83)

>>> 山田さんのお父さんへ

山田の成績は：

{'Math': 100, 'English': 60, 'Japanese': 83}

合計: 243

山田さんのお母さんへ

山田の成績は：

{'Math': 100, 'English': 60, 'Japanese': 83}

合計: 243

**Python関数のコメント方法**

Python公案に、「読みやすさは大切だ」と言っています。

Python関数には二つのコメント方法があります。

**方法一：**

**def 関数名(パラメータ1:'説明１', パラメータ2:'説明2') -> '戻り値の説明':**

**pass**

方法一の説明の部分はデータの型でもいいです。

例：

def foo(arg1, arg2):

return str(arg1) + arg2

help(foo)

>>> Help on function foo in module \_\_main\_\_:

foo(arg1, arg2)

def foo(arg1:'数字', arg2:'文字列') -> '文字列':

return str(arg1) + arg2

help(foo) # これでhelp()関数を使って説明内容が出ってきます。

>>> Help on function foo in module \_\_main\_\_:

foo(arg1:'数字', arg2:'文字列') -> '文字列'

def foo(arg1:int, arg2:str) -> str:

return str(arg1) + arg2

help(foo)

>>> Help on function foo in module \_\_main\_\_:

foo(arg1:int, arg2:str) -> str

**方法二：**

**def 関数名(パラメータ1, パラメータ2):**

**"""**

**docstring部分**

**説明内容：**

**...**

**"""**

**pass**

関数本体の先頭に文字列を組み込めば、関数定義にドキュメントを付けることができます。これを関数のdocstringと呼びます。

例：

def test(arg1, arg2):

"""A function for adding a number and a string"""

return str(arg1) + arg2

test(1, '3')

>>> '13'

help(test) # これでhelp()関数を使って説明内容が出ってきますよ。

>>> Help on function test in module \_\_main\_\_:

test(arg1, arg2)

A function for adding a number and a string

def test2(arg1:'数字', arg2:'文字列') -> '戻り値は文字列':

"""

第１引数は数字、第２引数は文字列

処理内容：

　　１．数字+文字列

　２．docstringの使い方を説明

"""

return str(arg1) + arg2

test2(2, '3')

>>> '23'

help(test2) # これでhelp()関数を使って説明内容が出ってきます。

>>> Help on function test2 in module \_\_main\_\_:

test2(arg1:'数字', arg2:'文字列') -> '戻り値は文字列'

第１引数は数字、第２引数は文字列

処理内容：

　　 １．数字+文字列

　 ２．docstringの使い方を説明

print(test2.\_\_doc\_\_) # ちなみに、関数のdocstring部分の内容は関数の\_\_doc\_\_にあります。

>>> 第１引数は数字、第２引数は文字列

処理内容：

　　 １．数字+文字列

　 ２．docstringの使い方を説明

# **再帰関数(recursive function)**

関数定義の中で、その関数自身を呼び出すことを「再帰呼び出し (recursive call) 」とか「再帰定義 (recursive definition) 」といいます。

再帰関数(recursive function)とは関数の中で自分自身の関数を呼び出す構造の関数です。

具体的なコード例をみてみましょう。

def sum(n): # 1からnまでの自然数の和を返す関数

if n == 0:

return n

return n + sum(n-1) # n! = n + n-1 + n-2 + n-3 + ... + 1 + 0

このように、自分自身を呼び出す処理が書かれている何かを呼び出すのが、再帰処理です。

sum(10)

>>> 55

**簡単な再帰関数の書き方**

**def 関数名(パラメータ名1, ...):**

**if 終了条件: 　 # 難しいところ1**

**return 終わる時の値**

**# TODO: 処理プロセス**

**return この関数(変化後の引数1, ...)を含む式 # 難しいところ2**

**※ 注意点：**

* １．終了条件が間違っていると永遠に処理が終わらない
* ２．入れ子が深くなると、メモリが足りなくなる

sum(10000000000) # 10000000000! 入れ子が深すぎる！

>>> --------------------------------------------------------------------------

RecursionError Traceback (most recent call last)

<ipython-input-7-427e65762c74> in <module>

---> 1 sum(10000000000) # 10000000000! 入れ子が深すぎる！

<ipython-input-5-b40679aa0df7> in sum(n)

2 if n == 0:

3 return n

---> 4 return n + sum(n-1) # n + n-1 + n-2 + n-3 + ... + 1 + 0

... last 1 frames repeated, from the frame below ...

<ipython-input-5-b40679aa0df7> in sum(n)

2 if n == 0:

3 return n

---> 4 return n + sum(n-1) # n + n-1 + n-2 + n-3 + ... + 1 + 0

RecursionError: maximum recursion depth exceeded in comparison

**エラーが出ますよ！**

def sum(n): # 1からnまでの偶数の和がほしいが...

if n == 0:

return n

return n + sum(n-2) # n + n-2 + n-4 + ... + 2 + 0 ?

sum(10) # 10 + 8 + 6 + 4 + 2 + 0 引数は偶数なら、大丈夫ですけど

>>> 30

sum(9) # 9 + 7 + 5 + 3 + 1 + -1 + ... 終了条件が間違っていると永遠に処理が終わらない

>>> ---------------------------------------------------------------------------

RecursionError Traceback (most recent call last)

<ipython-input-10-8886fa631239> in <module>

---> 1 sum(9)

<ipython-input-8-a63851d40caf> in sum(n)

2 if n == 0:

3 return n

---> 4 return n + sum(n-2) # n + n-2 + n-4 + ... + 2 + 0 ?

... last 1 frames repeated, from the frame below ...

<ipython-input-8-a63851d40caf> in sum(n)

2 if n == 0:

3 return n

---> 4 return n + sum(n-2) # n + n-2 + n-4 + ... + 2 + 0 ?

RecursionError: maximum recursion depth exceeded in comparison

**ループ(loop)に直す**

再帰という構造はループ(for文など)に変換することが可能と言われています。先程の例をループ(for文など)に直してみましょう。

def sum(n): # 1からnまでの自然数の和を返す関数

total = 0

for i in range(n + 1):

total += i # すなわち total = total + i

return total

sum(10) # 先と同じ結果

>>> 55

**再帰(recursive) vs ループ(loop)**

再帰呼出しと言っても、呼出しのための特別な構文があるわけではなく、普通に関数を呼出すだけです。呼出す側と呼出される側がたまたま同じ関数になっているだけのことです。

しかし、再帰呼出しには特別な魅力があります。何かの処理をくり返したいようなとき、for 文などのようなループを使うことが多いわけですが、そうした構文を使わずに再帰的な定義により計算することも可能です。実際、ある種の処理はくり返しを使うよりも再帰的に定義したほうが自然かつ簡潔に記述できることがあります。

フィボナッチ数の例を見てみましょう。

**フィボナッチ数(Fibonacci number)**

フィボナッチ数（Fibonacci number）は、イタリアの数学者レオナルド・フィボナッチにちなんで名付けられた数である。n番目のフィボナッチ数をFnで表すと、Fnは再帰的に

𝐹0=0,　𝐹1=1,　𝐹𝑛+2=𝐹𝑛+𝐹𝑛+1　(𝑛≧0)

で定義される。

すなわち[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, …]のような数列です。最初の二項は 0, 1 であり、以後どの項もその直前の2つの項の和となっている。

**Pythonでn番目のフィボナッチ数を求めよう**

方法一：

# ループ

def Fibonacci\_Loop(n):

a, b = 0, 1

for i in range(n):

a, b = b, a + b

return a

[Fibonacci\_Loop(x) for x in range(10)] # 最初の十項

>>> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

方法二：

# 再帰関数

def Fibonacci(n):

if n < 2:

return n

return Fibonacci(n-2) + Fibonacci(n-1)

[Fibonacci(x) for x in range(10)]　　　　 # 最初の十項

>>> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

上記の例のように、再帰的に定義したほうが自然かつ簡潔に記述できることがあります。

1. **エラー処理**

# **例外を投げる**

これまでのところ、私たちは多くのエラーに遭遇したでしょう。エラーには(少なくとも)二つのはっきり異なる種類があります。それは構文エラー(syntax error)と例外(exception)です。

構文エラーは構文解析エラーとしても知られており、Pythonを勉強している間に最もよく遭遇する問題の一つでしょう。すこしずつPythonの構文に慣れてよく練習すると、あるいは優れた開発ツールを利用すると、構文エラーを完全に回避することができます。

一方で、たとえ構文的に正しくても、実行しようとしたときにエラーが発生するかもしれません。実行中に検出されたエラーは例外と呼ばれ、常に致命的とは限りません。どんなに優秀なプログラマーが書いたコードにも、例外が発生することがあります。

コードの中にエラーが起きそうなところがあります。しかし、実際に実行してみないと、分かりません。この時、例外処理が必要です。エラーが起きそうなところに、すべて例外処理を追加して、ユーザに何が起きるのかを知らせておくのはいいプログラミング方法です。問題を解決できないかもしれませんが、少なくとも状況を知らせて穏便にプログラムを終了させることはできます。

プログラム内で例外処理を用意できていなければ、次のコードが示すように、Pythonはエラーメッセージとエラー発生箇所についての情報を出力し、プログラムを強制終了します。

small\_list = ['L', 'O', 'V', 'E'] # 四つの要素があります、indexは0から3まで

print(small\_list[5])

>>> ---------------------------------------------------------------------------

IndexError Traceback (most recent call last)

<ipython-input-24-58fde437b39a> in <module>

1 small\_list = ['L', 'O', 'V', 'E'] # 四つの要素があります、indexは0から3まで

---> 2 print(small\_list[5])

IndexError: list index out of range

エラーメッセージの最終行は何が起こったかを示しています。

例外は様々な型(type)で起こり、その型がエラーメッセージの一部として出力されます。上の例での型はIndexErrorです。例外型として出力される文字列は、発生した例外の組み込み名です。

**例外型一覧**

[item for item in dir(\_\_builtins\_\_) if item.endswith('Error')]

>>> ['ArithmeticError',

'AssertionError',

'AttributeError',

'BlockingIOError',

'BrokenPipeError',

'BufferError',

'ChildProcessError',

'ConnectionAbortedError',

'ConnectionError',

'ConnectionRefusedError',

'ConnectionResetError',

'EOFError',

'EnvironmentError',

'FileExistsError',

'FileNotFoundError',

'FloatingPointError',

'IOError',

'ImportError',

'IndentationError',

'IndexError',

'InterruptedError',

'IsADirectoryError',

'KeyError',

'LookupError',

'MemoryError',

'ModuleNotFoundError',

'NameError',

'NotADirectoryError',

'NotImplementedError',

'OSError',

'OverflowError',

'PermissionError',

'ProcessLookupError',

'RecursionError',

'ReferenceError',

'RuntimeError',

'SyntaxError',

'SystemError',

'TabError',

'TimeoutError',

'TypeError',

'UnboundLocalError',

'UnicodeDecodeError',

'UnicodeEncodeError',

'UnicodeError',

'UnicodeTranslateError',

'ValueError',

'WindowsError',

'ZeroDivisionError']

**raiseで例外を投げ出す**

上記のエラーメッセージの最終行はPythonのキーワードraiseを使って作られたものです。キーワードraiseを使用すると、故意に例外を発生させることが出来ます。

raiseの使い方：

**raise 例外型**

**# または**

**raise 例外型("この例外についての説明")**

例：

small\_list = ['L', 'O', 'V', 'E']

def Mylist(n):

if type(n) != int:

raise TypeError('indexには整数が必要です！')

elif n >= len(small\_list):

raise IndexError('リストの範囲に注意してください！')

else:

return small\_list[n]

Mylist('100')

>>> ---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-32-2ff2f825740f> in <module>

9 return small\_list[n]

10

---> 11 Mylist('100')

<ipython-input-32-2ff2f825740f> in Mylist(n)

3 def Mylist(n):

4 if type(n) != int:

---> 5 raise TypeError('indexには整数が必要です！')

6 elif n >= len(small\_list):

7 raise IndexError('リストの範囲に注意してください！')

TypeError: indexには整数が必要です！

# **try except**

**例外の捕捉**

例外を投げ出すだけで、プログラムは強制的に中止されます。もし例外が投げ出された後に、例外処理あるいは続いての問題のないコードを実行したい場合、例外の捕捉が必要になりました。

**try文で例外の捕捉と処理**

**# 他のコード(try文の外)**

**try:**

**エラーが起きそうなところのコード**

**except 例外型:**

**例外処理**

**# 他のコード(try文の外)**

try 文は下記のように動作します。

1. まず、try節(キーワードtryとexceptの間の文)が実行されます。
2. 何も例外が発生しなければ、 except節をスキップしてtry文の実行を終えます。
3. try節内の実行中に例外が発生すると、その節の残りは飛ばされます。次に、例外型がexceptキーワードの後に指定されている例外の型に一致する場合、except節が実行された後、 try文の後ろへ実行が継続されます。
4. もしも except節で指定された例外と一致しない例外が発生すると、その例外はtry文の外側に渡されます。例外に対する処理がどこにもなければ、処理されない例外となり、エラーメッセージを出して実行を停止します。

例：

small\_list = ['L', 'O', 'V', 'E']

try:

print(small\_list[4])

except IndexError as e: # ここのIndexError例外にeという名前をつける

print(e) # 出力：eの内容

print("ここには間違いがあって、私は対応して処理しました。")

print("順調に終わった！")

>>> list index out of range

ここには間違いがあって、私は対応して処理しました。

順調に終わった！

**※ 注意：具体的な例外の内容を利用したい場合、例外に名前をつける必要があります。〇〇ErrorはあくまでもPythonのクラスで、実際のオブジェクトではないので、オブジェクトとして操作できません。(Pythonのクラスについて、後記の章で説明致します)**

一つの try文に複数の except 節を設けて、さまざまな例外に対するハンドラを指定することができます。

例：

small\_list = ['L', 'O', 'V', 'E']

try:

print(small\_list[4])

except IndexError:

print("ここにはIndexErrorの間違いがあって、私は対応して処理しました。")

except TypeError:

print("ここにはTypeErrorの間違いがあって、私は対応して処理しました。")

except ValueError:

print("ここにはValueErrorの間違いがあって、私は対応して処理しました。")

print("順調に終わった！")

>>> ここにはIndexErrorの間違いがあって、私は対応して処理しました。

順調に終わった！

同時に一つ以上のハンドラ[[1]](#footnote-1)が実行されることはありません。ハンドラは対応するtry節内で発生した例外だけを処理し、同じtry文内の別の例外ハンドラで起きた例外は処理しません。except節には複数の例外を丸括弧で囲ったタプルにして渡すことができます。例えば以下のようにします:

try:

...

except (TypeError, IndexError, SyntaxError):

...

**Exceptionであらゆる例外を捕捉しましょう**

異なる型の例外に対して、異なるハンドラを行うのは良いプログラミング方法ですが、多くの場合、例外の型を予測することはできません。この時、すべての型の例外を表すExceptionを使います。

例：

small\_list = ['L', 'O', 'V', 'E']

try:

print(small\_list[4])

except Exception as e:

print(e)

print("ここには間違いがあって、私は対応して処理しました。")

print("順調に終わった！")

>>> list index out of range

ここには間違いがあって、私は対応して処理しました。

順調に終わった！

**else節：正常終了時の処理**

try節で例外が発生せず正常終了したあとに行う処理をelse節に指定できます。例外が発生してexceptでキャッチした場合はelse節の処理は実行されません。

例：

def divide(x, y):

try:

result = x / y

except ZeroDivisionError as e:

print("division by zero!")

else:

print("result is", result)

divide(10, 2) # 正常終了

>>> result is 5.0

**finally節：終了時に常に行う処理**

finally節は、例外が発生したかどうかに関わらず、try文を抜ける前に常に実行されます。try節の中で例外が発生して、 except節で処理されていない場合、またはexcept節かelse節で例外が発生した場合は、finally節を実行した後、その例外を再送出します。

例：

def divide(x, y):

try:

result = x / y

except ZeroDivisionError as e:

print("division by zero!") # エラーの時

else:

print("result is", result) # 正常の時

finally:

print("executing finally clause") # 最後

divide(10, 0) # 10 / 0

>>> division by zero!

executing finally clause

1. **入出力**

# **標準入出力**

この節では、よく使われるキーボードからの入力(標準入力)とディスプレイへの表示出力(標準出力)を紹介します。

**標準入力：input()**

input関数は(コマンドラインで)キーボードから入力した文字を取得するためのPythonの組み込み関数です。コマンドラインで何かと入力して、Enterを押すとinput関数を利用できます。

例：

a = input() # 変数aにinput関数で入力を受け付けた数値を代入して、print関数で表示しています。

print(a)

*(13を入力すると)*

>>> 13

**※ キーボードから何が入力されるにかかわらず、input関数は入力されたデータを文字列として扱います。戻り値は文字列(str)です!**

print(type(a))

>>> <class 'str'>

input関数の引数は入力に関する説明です。すなわち、input関数の引数は一つの文字列型(str)の変数です。そして、input関数の引数のデフォルト値は''です。

b = input('整数を入力してください:')

print(b)

print(type(b))

>>> 整数を入力してください: *(14を入力すると)*

14

<class 'str'>

入力した数値を数値として扱ってほしい場合、int()とfloat()関数で型を変換する必要があります。

c = int(input('数字を入力してください:'))

res = c \* 10

print(c)

print(type(c))

print(res)

>>> 数字を入力してください: *(14を入力すると)*

14

<class 'int'>

140

**標準出力：print()**

print関数を使用すると、文字列や変数の結果を簡単に出力することができます。print関数はプログラミングの初心者から上級者まで使う頻度が高いものです。(いままで、私たちは何度もprint関数を使用してきましたね。)この節で、print関数の使い方についてもっと詳しく説明します。

**1.print()関数の基本の使い方**

①文字列を出力する方法：

**print(‘出力したい文字列’)**

上記のように、()の中に任意の文字列を記入するだけで、出力が可能です。

例：

print('Hello World!') # これは私たちが最初に実現したPythonのプログラマですね。

>>> Hello World!

print('This', 'is', 'a', 'book.') # 複数の文字列をまとめて出力する

>>> This is a book.

②変数(リストや辞書の内容も)を出力する方法:

**print(変数1, 変数2, 変数3)**

　ご覧いただけるように、変数はコンマで区切ればいくつでも出力可能です。

例：

a = 9

b = "apples"

c = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print("I ate", a, b, c)

>>> I ate 9 apples [1, 2, 3, 4, 5, 6, 7, 8, 9]

上記のコードでは、aに整数9を、bに文字列apples、ｃにリストを格納し、それらを一緒に出力しました。

**２.さまざまなprint()関数の使い方**

**①複数の変数を文字列に埋め込んで出力：**

文字列の途中に変数の値を挿入して出力したい場合は以下の三つの方法があります。

* パーセント%を使う
* 文字列メソッドformat()を使う
* f文字列(フォーマット文字列)を使う

**a. パーセント%を使う**

以下のように書くと文字列中の変換指定子(%d, %f, %sなど)が変数で置換されます。変数が複数の場合はカンマ,で区切り()で囲んでタプルとして指定します。

**print(文字列 % (変数1, 変数2, ...))**

変換指定子は整数が%d、浮動小数点が%f、文字列が%s。よく使われる書式の指定方法は以下のテーブルを参照してください。

| **変換型[[2]](#footnote-2)** | **意味** |
| --- | --- |
| d | 符号付き10進整数 |
| i | 符号付き10進整数 |
| b | 符号付き2進数 |
| o | 符号付き8進数 |
| x | 符号付き16進数(小文字) |
| X | 符号付き16進数(大文字) |
| e | 指数表記の浮動小数点数(小文字) |
| E | 指数表記の浮動小数点数(大文字) |
| f | 10進浮動小数点数 |
| F | 10進浮動小数点数 |
| c | 文字一文字 |
| r | 文字列(repr()で変換) |
| s | 文字列(str()で変換) |

例：

age = 23

print('私は今年%d歳です。' % age) # 一つの変数の場合

>>> 私は今年23歳です。

name = '黒岩晶'

age = 18

print('%sさんは今年%d歳になりました。' % (name, age))

>>> 黒岩晶さんは今年18歳になりました。

name = '黒岩晶'

score = 99.5

print('%sさんの成績は%f点です。' % (name, score))

>>> 黒岩晶さんの成績は99.500000点です。

※ 注意：異なる変換指定子によって、同じ変数を利用して出力しても、異なる値になるかもしれません。

number = 122

print('%s, %d, %f, %c, %e, %x, %X' % (number, number, number, number, number, number, number))

>>> 122, 122, 122.000000, z, 1.220000e+02, 7a, 7A

無理やり変換指定子を使うと、エラーが出る可能性もあります。基本的に整数が%d、浮動小数点が%f、文字列が%sを暗記するだけでは十分です。

figure = 3.1415926

print('%c' % figure) # %c:一文字

>>> ---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-31-e28826161893> in <module>

1 figure = 3.1415926

---> 2 print('%c' % figure)

TypeError: %c requires int or char

**b. format()魔法(メソッド)を使う**

異なる型の変数に異なる変換指定子を指定するのはちょっと面倒くさいです。Pythonにはもっと優しい方法があります。Pythonの文字列メソッドformat( )を使って以下のように書くと文字列中の置換フィールド{ }が引数に指定した変数で置換されます。変数が複数の場合はカンマ,で区切ります。変換指定子を指定しなくても大丈夫です。

**print(文字列.format(変数1, 変数2, ... ))**

例：

name = '黒岩'

age = 23

print('{}は今年{}歳になりました。'.format(name, age))

>>> 黒岩は今年23歳になりました。やっと23歳のいい大人になりました！

置換フィールド{ }に文字列を指定すると、キーワード引数または辞書型として指定した値でも置換されます。

例：

print('{name}は今年{age}歳になりました。'.format(name='黒岩', age=23))

>>> 黒岩は今年23歳になりました。

a = '黒岩'

b = 23

print('{name}は今年{age}歳になりました。'.format(name=a, age=b))

>>> 黒岩は今年23歳になりました。

#「\*\*辞書型」イコール「key1＝value1, key2=value2, ...」

print('{name}は今年{age}歳になりました。'.format(\*\*{'name':'黒岩','age':23}))

>>> 黒岩は今年23歳になりました

output = {'name':'黒岩','age':23}

print('{name}は今年{age}歳になりました。'.format(\*\*output))

print('{0[name]}は今年{0[age]}歳になりました。'.format(output)) # 0: output

>>> 黒岩は今年23歳になりました。

黒岩は今年23歳になりました。

※ 注意：辞書型オブジェクトを置換フィールドに指定する場合、文字列メソッドformat()ではキーの指定に引用符'と "は不要。引用符をつけるとエラーになります。

output = {'name':'黒岩','age':23}

print('{0["name"]}は今年{0["age"]}歳になりました。'.format(output)) # 0: output

>>> ---------------------------------------------------------------------------

KeyError Traceback (most recent call last)

<ipython-input-74-4e21d1218173> in <module>

1 output = {'name':'黒岩','age':23}

---> 2 print('{0["name"]}は今年{0["age"]}歳になりました。'.format(output)) # 0: output

KeyError: '"name"'

文字列中の波括弧{ }をそのまま出力したい場合は{{ }}とします。

例：

print('{name}は今年{age}歳になりました。{{いい大人になりますよね！}}'.format(name='黒岩', age=23))

>>> 黒岩は今年23歳になりました。{いい大人になりますよね！}

**formatメソッドで書式指定の基本方法：**

formatメソッドは単に文字列を埋め込めるだけでなく、オプションを指定すれば、文字列を整形して埋め込むことができます。オプションは{ }中の後方にコロン:を付けて指定します。

**{引数のインデックス:オプション}**

**#あるいは**

**{:オプション}**

**オプション:**

**ⅰ. 数値表現**

パーセント%を使う時と同じように、formatメソッドも特な変換型を指定することができます。変換型は「%を使う」部分の内容を参考してください。

例：

number = 3

figure = 0.1415926

print('浮動小数点数{1:f}と整数{0:d}'.format(number, figure)) # ここのインデックスの使い方に注意

>>> 浮動小数点数0.141593と整数3

number = -123

print('{0}は2進数だと{0:b}、8進数だと{0:o}、16進数だと{0:X}'.format(number))

>>> -123は2進数だと-1111011、8進数だと-173、16進数だと-7B

**ⅱ. 桁数指定（精度）**

小数点以下?桁を指定するには、{インデックス: .?f}のようにピリオド.の後に整数で桁数を指定します。

例：

figure = 0.1415926

print('浮動小数点数{0:.3f}'.format(figure))

>>> 浮動小数点数0.142

print('浮動小数点数{0:.4f}'.format(figure))

>>> 浮動小数点数0.1416

また、この桁数指定を文字列タイプに使用すると({インデックス: .?s}あるいは{インデックス: .?})、文字数を最大何文字使用するかを指定できます。

例：

information = '黒岩 晶'

print('苗字は{:.2}'.format(information))

>>> 苗字は黒岩

**ⅲ. 幅**

formatメソッドで変数を文字列に埋め込む際には、埋め込む際の最大幅と表示位置を指定することもできます。表示する最大幅は、桁数指定の前で指定します。最大幅は?の指定方法は{ :?}です。ミスをしやすいので、最大幅であることに留意してください。

例：

information = 'abcdefghi'

print('情報{:20}'.format(information)) # 文字列の最大幅(デフォルト左詰め)

>>> 情報abcdefghi

figure = 1233123.1345663

print('浮動小数点数{0:15.4f}'.format(figure)) # 浮動小数点数の最大幅 + .桁数指定(デフォルト右詰め)

>>> 浮動小数点数 1233123.1346

実は、指定された最大幅を超えても問題ないです。最大幅は主に表示位置を指定するために使います。

| **型名** | **説明** |
| --- | --- |
| <最大幅 | 要素を左詰めし、残りの幅をスペースで埋める |
| >最大幅 | 要素を右詰めし、残りの幅をスペースで埋める |
| ^最大幅 | 要素を中央寄せし、残りの幅をスペースで埋める |

例：

a = '左詰め'

b = '右詰め'

c = '中央寄せ'

print('{:<20}'.format(a)) # 最大表示幅：20

print('{:>20}'.format(b))

print('{:^20}'.format(c))

>>> 左詰め

右詰め

中央寄せ

幅はあらかじめスペースキーを使って入れておいてもいいですが、数字で指定するほうがパッと見てどれだけの幅を取っているかが分かりますし、ソースコードもスッキリします。

**ⅳ. 他のオプション**

　すべての情況に満足できるようにformatメソッドにはもっと複雑な書式指定がたくさんあります。詳しくはPython 3.6.5ドキュメントを参考してください。

**c. f文字列(フォーマット文字列)を使う**

　f文字列は文字列リテラルの前にfをつけた文字列です。

**f'xxx'**

文字列中の置換フィールド{ }に変数を直接指定できます。

例：

name = '黒岩'

age = 23

print(f'{name}は今年{age}歳になりました。')

>>> 黒岩は今年23歳になりました。

output = {'name':'黒岩', 'age':23}

print(f"{output['name']}は今年{output['age']}歳になりました。") # 辞書型の変数を利用して出力

>>> 黒岩は今年23歳になりました。

※ 注意：辞書型オブジェクトを置換フィールドに指定する場合、文字列メソッドformat()にひきかえ、f文字列では置換フィールドがそのまま式として評価されるので引用符が必要です。引用符が無いとエラーになります。

output = {'name':'黒岩', 'age':23}

print(f"{output[name]}は今年{output[age]}歳になりました。") # nameとage引用符が無い

>>> ---------------------------------------------------------------------------

KeyError Traceback (most recent call last)

<ipython-input-75-7c0141ae80be> in <module>

1 output = {'name':'黒岩', 'age':23}

---> 2 print(f"{output[name]}は今年{output[age]}歳になりました。")

KeyError: '黒岩'

上記の例のように、f文字列で出力するのは一番簡潔な方法です。

f文字列で書式指定の方法はformatメソッドと同じです。

例：

name = '黒岩　晶'

score = 99.25

# nameは最大２文字を使用し、scoreは小数点以下１桁を指定する

print(f'{name:.2}の成績は{score:.1f}。')

>>> 黒岩の成績は99.2。

figure = [1234.2312, 23.3222231, 45.3131, 3122315.4541, 4352.1, 0.23134]

for item in figure:

print(f'右詰め浮動小数点数(小数点以下2桁を指定する):{item:>20.2f}')

>>> 右詰め浮動小数点数(小数点以下2桁を指定する): 1234.23

右詰め浮動小数点数(小数点以下2桁を指定する): 23.32

右詰め浮動小数点数(小数点以下2桁を指定する): 45.31

右詰め浮動小数点数(小数点以下2桁を指定する): 3122315.45

右詰め浮動小数点数(小数点以下2桁を指定する): 4352.10

右詰め浮動小数点数(小数点以下2桁を指定する): 0.23

他のオプションは前の「ⅳ.他のオプション」セクションに参考してください。

**②ほかのprint()関数の使い方：**

help(print)

>>> Help on built-in function print in module builtins:

print(...)

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

各引数の意味合いはそれぞれ次のとおりです。

* value: 出力対象のオブジェクト(上記のように文字列、変数など)。複数個渡すことができる。
* file: 出力先(ファイルなど)。デフォルトは標準出力だが、ファイルオブジェクトなどを指定することもできる。
* sep: valueが複数個渡された場合の区切り文字（セパレータ）。デフォルトは半角空白。
* end: 最後の要素の末尾に付けられる文字。デフォルトは改行文字(\n)。
* flush: バッファなしで出力するかどうか。デフォルトは False で、出力先によって自動的に定められる。

# 通常最後に追加されるのは改行文字(\n)です。

print('土曜日')

print('日曜日')

>>> 土曜日

日曜日

# 通常最後に追加される改行を''に変換します。

print('土曜日', end='')

print('日曜日')

>>> 土曜日日曜日

# 通常最後に追加される改行を'、'に変換します。

print('土曜日', end='、')

print('日曜日')

>>> 土曜日、日曜日

# 通常半角空白で複数の出力対象を区切ります。

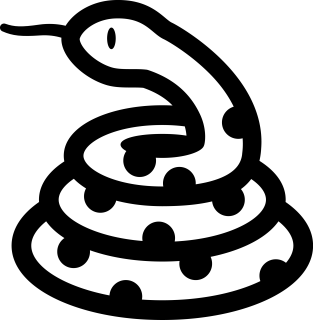
print('月曜日', '火曜日', '水曜日', '木曜日', '金曜日', '土曜日', '日曜日')

>>> 月曜日 火曜日 水曜日 木曜日 金曜日 土曜日 日曜日

# ','で複数の出力対象を区切ります。

print('月曜日', '火曜日', '水曜日', '木曜日', '金曜日', '土曜日', '日曜日', sep=',')

>>> 月曜日,火曜日,水曜日,木曜日,金曜日,土曜日,日曜日

****

**第三章**

**終日乾乾す**

1. **オブジェクト指向プログラミング**

# **クラスとオブジェクト**

**6.1.1関数型プログラミング(Functional Programming)**

これまでのところ、Pythonでプログラムを書くことについてさまざまな基礎知識を紹介いたしました。とくに3番目の節に、関数について紹介しました。これらの知識を用いて、すでにさまざまな機能のプログラムを作成することができます。さまざまな関数を使用してプログラムを書くこの方法は関数型プログラミングと呼ばれています。関数型プログラミングが注目されて久しいです。実際、数十年前にすべてのプログラムはこの方法で書かれたものでした。関数型プログラミングの考え方は、「データに何らかの処理を加えていく」の連続で組み立てていくものです。プログラムの関数と言うより数学の関数をイメージするといいかもしれません。

…⇒データA ⇒ 関数foo1(データA) ⇒ データA' ⇒ 関数foo2(データA') ⇒ データA''⇒…

理論的には、関数型プログラミングを用いて、任意の計算問題を解決することができます。最も有名なプログラミング言語のC言語は良い例です。C言語は典型的な関数型プログラミング言語です。また、C言語で任意のロジックのプログラムを実装することができます。有名なWindows、LinuxとMacシステムもC言語とC++言語で書かれたものです。さらに、Python言語自体もC言語で書かれたスクリプト言語の一つです。初期の頃、関数型プログラミングがプログラミングの主流となりました。しかし、プログラム内のコードの量が増えるにつれて、関数型プログラミングの欠点はますます明白になります。以下の例を見てみてください。

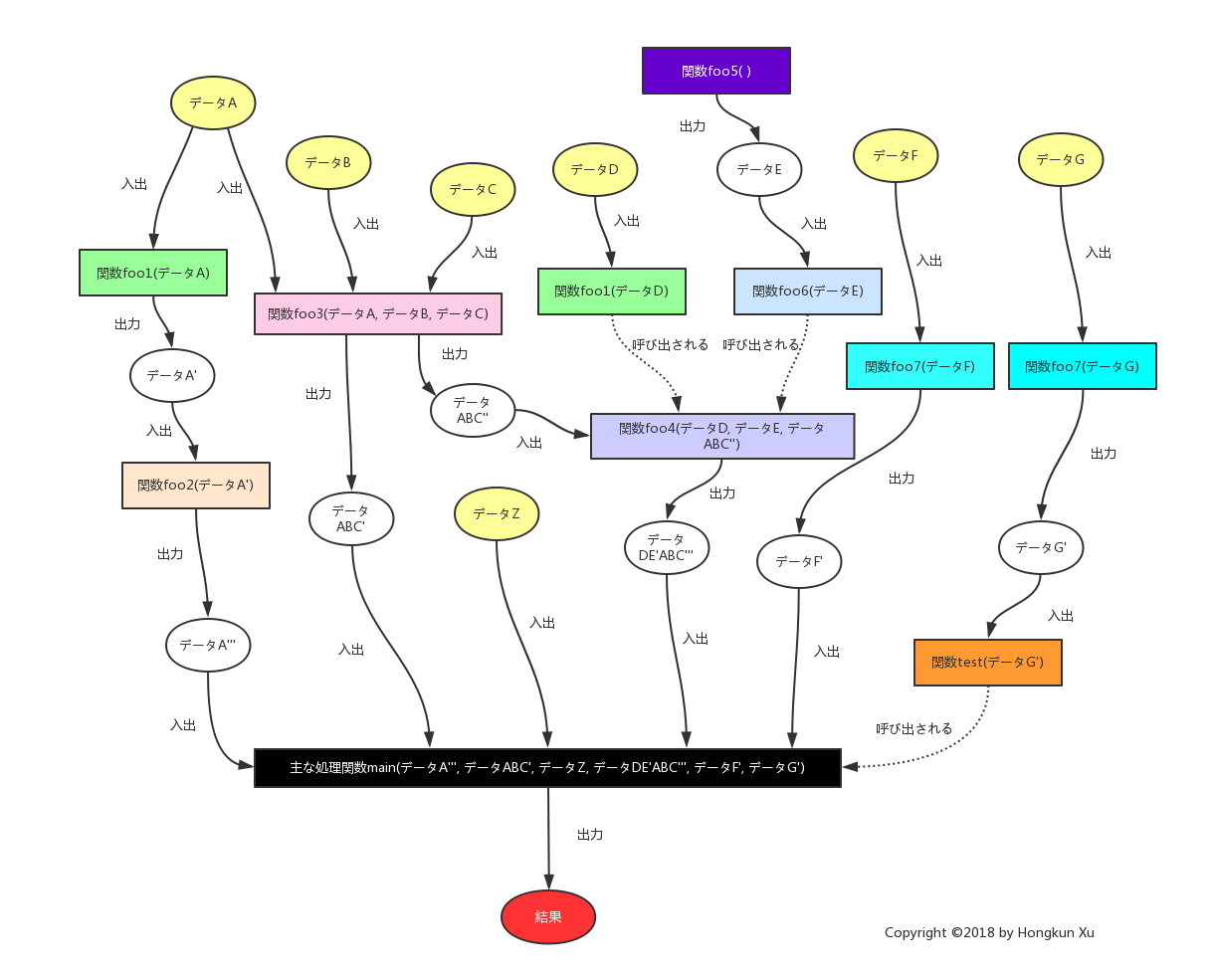


図6.1関数型プログラミング

この図を見ると、関数型プログラミングの最大の欠点は、複雑でメンテナンスが非常に困難であることが理解するでしょう。上記の図では、いくつかの関数とデータが複数回使用されること、一部の関数の出力値は、他の関数によって呼び出されることや、一部の関数は、データの初期化にのみ使用されることなどさまざまな情況があります。実際には、上記の図には九つの関数しかありません。何百もの関数を使用してプログラムを作成するときに、関数間の関係がどのように複雑で、変更がどのように困難で、メンテナンスがどのように難しいのか、想像するだに恐ろしいです。さらに、関数型プログラミングには多くの制限がありますが、ここでは詳しく説明しません。

**6.1.2オブジェクト指向プログラミング(Object Oriented Programming)**

関数型プログラミングの欠点を埋め合わせるには、物事に対する理解方法を見直す必要があります。データはスタティック(static)なものであり、関数はスタティックなデータを処理するプロセスです。単純にプログラムがスタティックなデータとそれを処理する関数だけで構成されているものだと考えると、関数型プログラミングになります。実際には、データと関数の間には関連性があります。関数は特定の構造のデータのみを処理でき、同じ構造のデータは、特定の一つまたは複数の関数によって処理されることができます。さらに、データも単独で存在せず、特定の物事を記述するために使用されます。たとえば、ヒトの名前、年齢、身長、体重はすべて、ヒトの属性を表すために使用される属性あるいはアトリビュート(attribute)です。そして、名前を変えるとか、名前をほかの人に教えるとか、痩せるとか、太るとか、これらはすべて、ヒトのアトリビュート(attribute)を操るメソッドです。アトリビュートとメソッドと一緒にヒトの分類を構成します。もっと簡単に説明すれば、データは人の属性(名詞)、関数は人のアクション(動詞)と考えることができます。下の図を見てください。

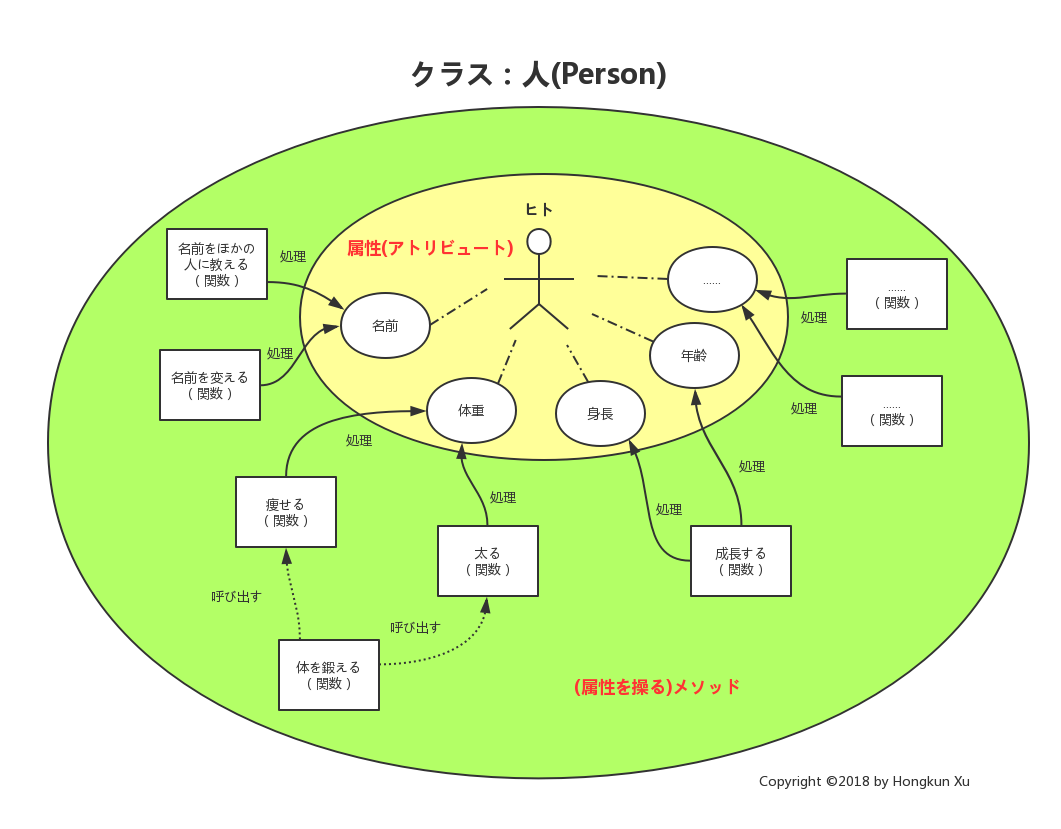


図6.2クラス(Class):ヒト

上図のさまざまな関係を省略すると、結果は次の図のようになります。

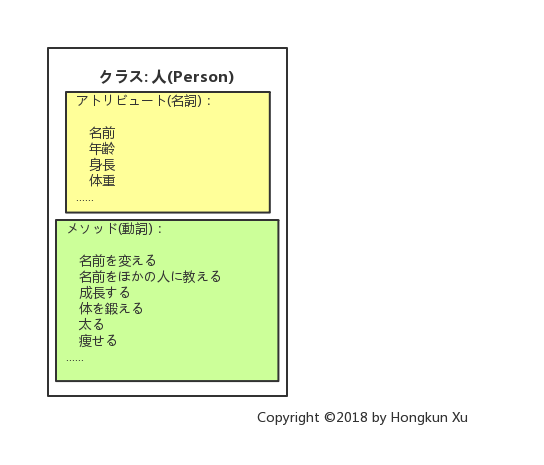


図6.3クラス(Class):ヒト(簡略図)

このように、我々は抽象的な種類(クラス：ヒト)を定義することに成功しました。オブジェクト指向プログラミングでは、ここの種類を直接クラス(Class)と呼びます。概念的にはクラスはオブジェクト(object)あるいは物事の種類を表します。このためオブジェクトはクラスに属するという言い方をします。あるクラスに属するオブジェクトのことをそのクラスのインスタンス(instance)と呼びます。データ型の理論から見た場合クラスは型を定義する手段の一つでもあります。クラスによってオブジェクトを記述する言語をクラスベース (class-based) のオブジェクト指向プログラミング言語と呼びます。Pythonはオブジェクト指向プログラミング言語の中の一つです。下図を見てください。鈴木さん、立花さん、山下さん、木村さんなどのオブジェクトはヒトのクラスのインスタンスです。

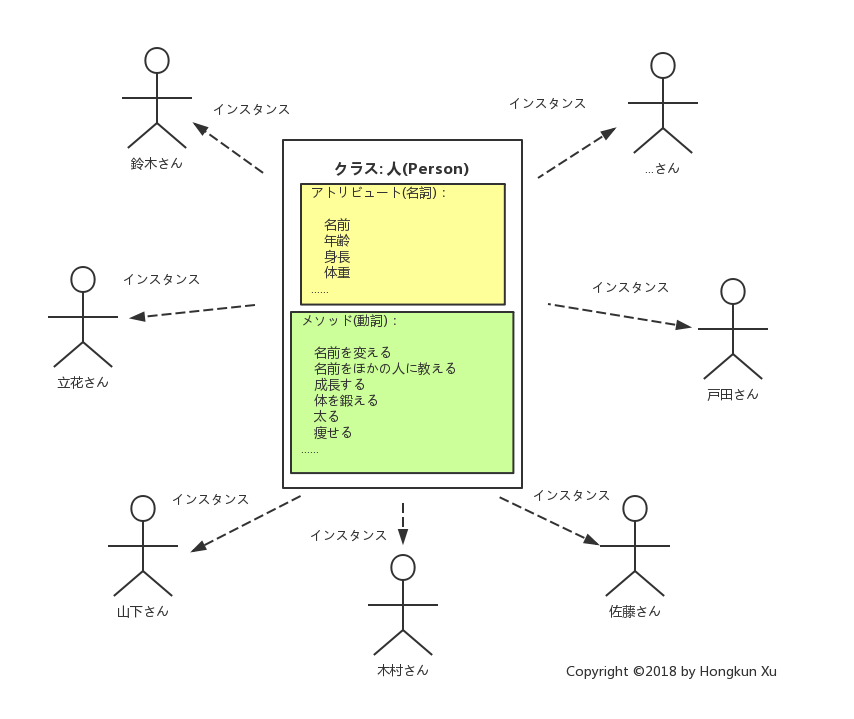


図6.4インスタンス(オブジェクト)

　クラスの定義に新しいアトリビュートまたはメソッドを追加すると、このクラスのすべてのインスタンスにもこの新しいアトリビュートまたはメソッドが追加されます。たとえば、ヒトのクラスの定義に"性別"を追加すると、すべてのヒトのクラスのインスタンスに"性別"というアトリビュートが追加されます。オブジェクト指向プログラミングでアトリビュートまたはメソッドの変更はとても便利です。もちろん、ほかのプログラミング手法もありますが、今一番流行っているプログラミング手法はやっぱりオブジェクト指向プログラミングです。

**6.1.3 classによるクラスの定義**

Pythonではクラスはclass クラス名:で作成します。クラス名の先頭は大文字の方がいいです。Pythonの関数の定義と同じように{ と }がありません。Pythonは同じ数の空白でインデントされたまとまりを一つのクラス(ブロック)と認識します。1レベルインデントするごとに、スペース(半角)を4つ使いましょう。そして、アトリビュートとメソッドは字下げを使用して定義します。ただし、Pythonではクラスのメソッドは最低1つの引数を持ちます。メソッドの最初の引数は必ずselfという名前にする慣例があります。selfを使用することで、オブジェクト自身のアトリビュートを取得したり、メソッドを呼び出すことが出来ます。引数なしの場合、オブジェクトはオブジェクト名 = クラス名( )で生成します。メソッドはオブジェクト名.メソッド名( )で呼び出します。

最初は考えられる限りもっとも単純なクラス、空クラスから始めましょう。

class Person():

pass # pass：何もしない

　オブジェクトは、クラス名をまるで関数のように呼び出して作ります。

someone = Person() # オブジェクト

この場合、Person( )は、Personクラスから１個のオブジェクトを作ります。しかし、Personクラスは空なので、そこから作ったsomeoneオブジェクトはただ存在するというだけで、ほかに何もできません。実際にこのようなクラスを定義することはないでしょう。

次は、クラスの中に一つのメソッドを定義しましょう。

class Person():

def say(self, name): # メソッドの最初の引数は必ずselfという名前の引数を持ちます

print("私の名前は", name, "です。", sep='')

someone = Person() # オブジェクト

someone.say('松本佐倉') # メソッドを呼び出します

>>> 私の名前は松本佐倉です。

**※ メソッドはオブジェクト名.メソッド名( )で呼び出します。**

**6.1.4 Pythonのコンストラクタ(Constructor): \_\_init\_\_メソッド**

今度は、Pythonオブジェクトを初期化する特殊メソッド\_\_init\_\_を含む形でもう一度クラスを定義しましょう。

class Person():

def \_\_init\_\_(self, name):

self.name = name

def say(self):

print("私の名前は", self.name, "です。", sep='') # 出力はself.nameに書き直しますよ!

これは、実際のPythonクラス定義に含まれているものです。\_\_init\_\_( )は見慣れない感じでしょう。\_\_init\_\_( )は、クラス定義から個々のオブジェクトを作るときにそれを初期化するメソッドに付けられた特殊名です。このように先頭と末尾にダブルアンダースコアをもっている特別なメソッドに触れたのは初めてですね。しかし、実際、これまで無意識の内に我々は多くの特別なメソッドを使用してきました[[3]](#footnote-3)。簡単に言えば、先頭と末尾にダブルアンダースコアを持っているメソッドは特定な目的を達成するための特殊名を付けられたメソッドです。こちらの特別なメソッドこそがPythonオブジェクトのさまざまな特性を育みます。(**マジックメソッド**とも呼ばれます。)

ほかに、self引数は、作られたオブジェクト自体を参照することを示します。これから、selfをクラスのインスタンスとして考えたほうがいいです。そして、初期化メソッド\_\_init\_\_( )にnameという引数を追加しました。

今度はこのようにオブジェクトを初期化します。name引数として文字列を渡せば、Personクラスからオブジェクトを作れるようになりました。クラスの初期化の引数は即ち\_\_init\_\_メソッドの引数です。

someone = Person('松本佐倉')

このコード行が行うことは、下図の通りです。

1. Personクラスの定義を探します。
2. メモリ内に新しいオブジェクトのインスタンスを作成します。
3. 新しく作ったオブジェクトをself、'松本佐倉'をnameとして渡して、オブジェクトの\_\_init\_\_メソッドを呼び出します。
4. nameの値をオブジェクトに格納します。
5. この新しいオブジェクトを返します。
6. オブジェクトにsomeoneという名前を与えます。

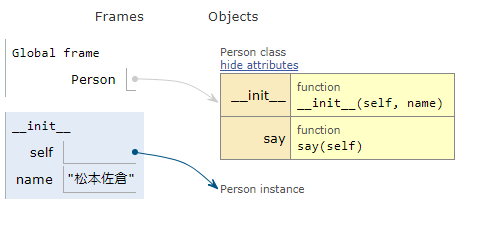


図6.5クラスPersonのインスタンスを作る瞬間の動き

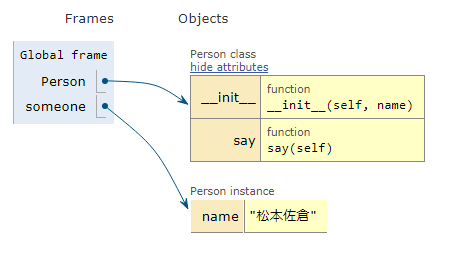


図6.6クラスPersonのインスタンスsomeoneを作った結果

someone.say() # 引数なし

>>> 私の名前は松本佐倉です。

なお、Personクラスの内部では、nameアトリビュートにはself.nameという形でアクセスします。だから、sayメソッドには引数もないのに、nameアトリビュートが操作できます。self.nameのようなアトリビュートは、Personクラスの内部ではどこでも直接に操作できます。オブジェクトの外からはsomeone.nameの形でも呼べます。

**※ これから、selfをクラスのインスタンスとして考えたほうがいいです。**

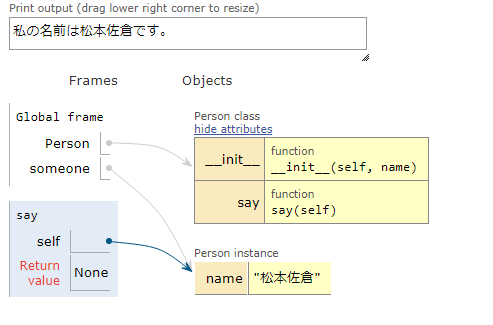


図6.7オブジェクトsomeoneのsayメソッドを呼ぶ

print(someone.name)

>>> 松本佐倉

他のオブジェクト指向プログラミング言語の定義を参照すると、\_\_init\_\_メソッドはクラスのコンストラクタ(Constructor)[[4]](#footnote-4)と呼ばれることがよくあります。

**豆知識:オブジェクトを初期化するとき、はじめて実行されるメソッドは\_\_init\_\_()メソッドではない**

Pythonでは、\_\_new\_\_メソッド、\_\_init\_\_メソッドの2つのメソッドを実装したクラスを生成することで、そのクラスのインスタンス作成時に行う処理を定義することができます。最初に実行されるのは\_\_new\_\_メソッド、次は\_\_init\_\_メソッドです。

class A():

def \_\_new\_\_(cls):

print('I am \_\_new\_\_メソッド')

instance = super().\_\_new\_\_(cls)

return instance

def \_\_init\_\_(self):

print('I am \_\_init\_\_メソッド')

a = A()

>>> I am \_\_new\_\_メソッド

I am \_\_init\_\_メソッド

上記の例を見るに、\_\_new\_\_メソッドと\_\_init\_\_メソッドはどちらもインスタンス生成時に呼ばれるため、初期化処理などはどちらに書いてもよいように思えます。しかし、これはおお間違えです。では、この2つのメソッドの違いは一体なんでしょうか。

実はインスタンス生成時に必ず呼ばれる\_\_new\_\_メソッドと違い、\_\_init\_\_メソッドはクラスの実装次第では呼ばれないこともあります。下記の例を見てください。

class A():

def \_\_new\_\_(cls):

print('I am \_\_new\_\_メソッド')

instance = super().\_\_new\_\_(cls)

# return instance

def \_\_init\_\_(self):

print('I am \_\_init\_\_メソッド')

a = A()

>>> I am \_\_new\_\_メソッド

\_\_init\_\_メソッドのself引数はクラスのインスタンスでしょう。実は、\_\_init\_\_メソッドのself引数(クラスのインスタンス)は\_\_new\_\_メソッドによって作成および返されたものです。\_\_new\_\_メソッドのcls引数はこのクラス自体です。そして、\_\_new\_\_メソッドの戻り値はこのクラスの一つのインスタンス(さまざまなメソッドに利用されるself引数の値)です。\_\_init\_\_メソッドの機能は、単にインスタンスを初期化することです。本当にインスタンスを作るのは\_\_new\_\_メソッドです。それで、もし\_\_new\_\_メソッドは値を返さない場合、実際のインスタンスは使いものになりません。このクラスのほかのすべてのメソッドが使えなくなります。

Python言語は、C++などのオブジェクト指向プログラミング言語とは異なり、完全にオブジェクト指向のプログラミング言語です。Javaもよく完全にオブジェクト指向の言語と言われていますが、PythonはJava言語を超えて、関数、クラス自体までもobjectというメタクラス(Metaclass)のインスタンスです。言い換えれば、Pythonのすべてがオブジェクトに違いない。\_\_new\_\_メソッドはメタクラスに関するメソッドです。普通に使う場合がほとんどないですが、ライブラリやAPIなどのプロジェクトの作成によく使います。

**6.1.5 Pythonのデストラクタ(Destructor):\_\_del\_\_メソッド**

Pythonクラスの\_\_del\_\_メソッドはインスタンスが破棄されるときに呼び出されます。これはファイナライザ(Finalizer)や (適切ではありませんが) デストラクタ(Destructor)とも呼ばれます[[5]](#footnote-5)。\_\_del\_\_メソッドを紹介する前に、まずPythonのガベージコレクションメカニズム(Garbage Collection Mechanism)の基本を説明いたします。下記の例を見てください。

例：

a = [1, 2, 3]

b = a

c = b

print(a, b, c)

a = [0, 0, 0]

print(a, b, c)

del b

del c

print('finish')

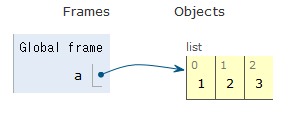
>>> [1, 2, 3] [1, 2, 3] [1, 2, 3]

[0, 0, 0] [1, 2, 3] [1, 2, 3]

finish

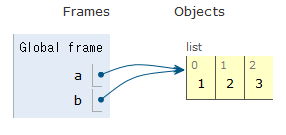
上記のdel文は引数に指定したオブジェクトの参照を削除するための文です。\_\_del\_\_メソッドと直接の関係を持っていません。では、実際のPythonの動きを見ながら、理解しましょう。

a = [1, 2, 3]



前の説明のように、Pythonのリストもクラスで、この行のコードは一つのリストのインスタンスを作って、そのインスタンスのラベル(label)をaと名前付けます。言い換えれば、変数aの参照先はこの新しい作られたリストのオブジェクトです。Pythonでは、参照カウンタ(reference counter)というものがあります。オブジェクトが作られたとき、そのオブジェクトの参照カウントは0です。オブジェクトがある変数に割り当てられたとき、参照カウントが1増やされます。したがって、この時、このリストのインスタンスの参照カウンタは１になりました。

b = a

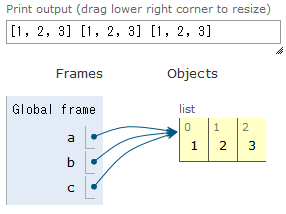


次に、変数bも変数aの参照先であるオブジェクトを参照します。この時、このリストのインスタンスの参照カウンタは2になりました。そして、変数aと変数bは同じオブジェクトを参照するので、このオブジェクトのアトリビュートを変えれば、変数aも変数bもそれによって一緒に変わります。

c = b

print(a, b, c)

>>> [1, 2, 3] [1, 2, 3] [1, 2, 3]

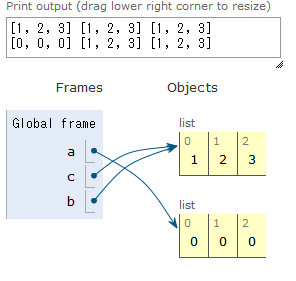


次に、変数cも変数bの参照先である変数aの参照先のオブジェクトを参照します。このリストのインスタンスの参照カウンタは3になりました。さきと同じように、このオブジェクトのアトリビュートを変えれば、三つの変数も一緒に変わります。

a = [0, 0, 0]

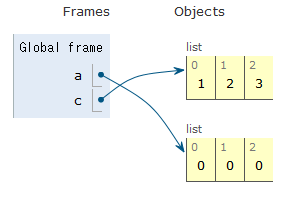
print(a, b, c)

>>> [0, 0, 0] [1, 2, 3] [1, 2, 3]



このコードによって、新たなリストのインスタンスを作りました。変数aに新たなオブジェクトそのものが代入されるときには、これまでの参照先から変わって新たなオブジェクトが作られた場所を参照するようになります。変数がスコープから外れたり、その変数に別の値が代入されたり、明示的にdelされたりすると、変数にもともと割り当てられていたオブジェクトの参照カウントは1減らされます。従って、この時、新たなリストのインスタンスの参照カウントは１になると同時に、元のリストのインスタンスの参照カウントは2になります。

del b

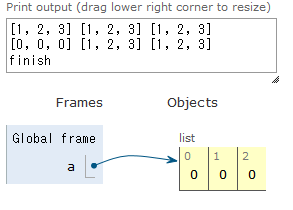


次に、del文によって、変数b(オブジェクトに対するbというラベル)を削除しました。この時、二つのインスタンスの参照カウントも1です。

del c

print('finish')

>>> finish



最後に、del文によって、変数c(オブジェクトに対するcというラベル)を削除しました。この時、最初のインスタンスの参照カウントは0になります。Pythonでは、参照カウントが0になったオブジェクトはどこからも参照されていないとみなされ、解放されます。したがって、最後に変数aしかありません。

上記の参照カウントの動きやオブジェクトの解放などのコントロールはPythonのガベージコレクタ(Garbage Collector)で自動的に実施されます。これこそが、Pythonのガベージコレクションメカニズム(Garbage Collection Mechanism)の基本です[[6]](#footnote-6)。

では、\_\_del\_\_メソッドは一体どの瞬間に呼び出されるのでしょうか。答えは、参照カウントは1から0になったときです。次の例を見てください。

例：

class Person():

def \_\_init\_\_(self, name):

self.name = name

def \_\_del\_\_(self):

print('\_\_del\_\_()メソッドを実行する')

a = Person('黒岩')

b = a

c = b

print(a)

print(b)

print(c)

del a

del b

del c # 参照カウントは1から0になるとき、ここで\_\_del\_\_が呼び出される

>>> <\_\_main\_\_.Person object at 0x000000000505A358>

<\_\_main\_\_.Person object at 0x000000000505A358>

<\_\_main\_\_.Person object at 0x000000000505A358>

\_\_del\_\_()メソッドを実行する

出力の結果を見ると、参照カウントは1から0になったとき、\_\_del\_\_メソッドはPythonのガベージコレクタによって自動的に呼び出されることがわかるようになるでしょう。Pythonの実際の動きは下の図のようになります。

a = Person('黒岩')

b = a

c = b

print(a)

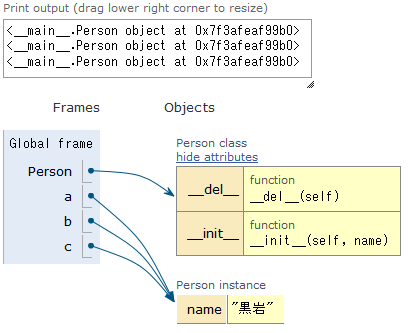
print(b)

print(c)

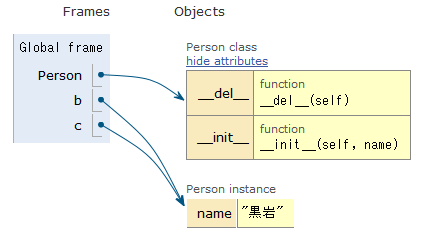
>>> <\_\_main\_\_.Person object at 0x7f3afeaf99b0>

<\_\_main\_\_.Person object at 0x7f3afeaf99b0>

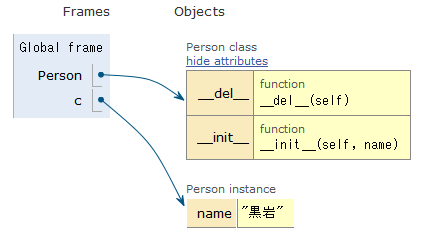
<\_\_main\_\_.Person object at 0x7f3afeaf99b0>



del a

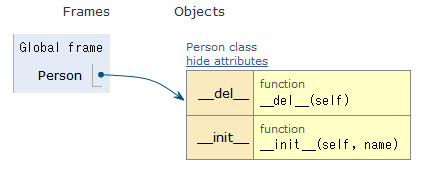


del b



del c # 参照カウントは1から0になるとき、ここで\_\_del\_\_が呼び出される

>>> \_\_del\_\_()メソッドを実行する



**\_\_del\_\_の落とし穴**

ほとんどの場合、\_\_del\_\_メソッドを自分で定義する必要がありません。Pythonの自動ガベージコレクションメカニズムは何度も改良されており、すでに優れたパフォーマンスを持っています。そして、\_\_del\_\_メソッドはとても使いにくいメソッドで、無意識のうちに非常に深刻な副作用を引き起こす恐れがあります。なぜなら、Pythonのインタプリタが終了したときに、残存しているオブジェクトの\_\_del\_\_メソッドが呼び出される保証はありません[[7]](#footnote-7)。したがって、\_\_del\_\_メソッドを自分で定義するのは非常に賢明ではない行動です。「クラスで\_\_del\_\_メソッドを定義しない」、基本的にはこれがいちばん安全です。\_\_del\_\_メソッドがないのであれば、プログラマは基本的にメモリリークの心配をする必要はありません。

**6.1.6あるオブジェクトが特定のクラスのインスタンスかチェックしよう**

多くの場合、あるラベル(一つの英語単語)がインスタンス名またはクラス名を表しているかどうかを見分けるのは困難です。または、オブジェクトがどのクラスのインスタンスであるかを正確に見分けるのが難しい場合もあります。このとき、type関数あるいはisinstance関数を使いましょう。

例：

class Person():

def \_\_init\_\_(self, name):

self.name = name

def say(self):

print("私の名前は", self.name, "です。", sep='') # 出力はself.nameに書き直しますよ!

someone = Person('松本佐倉')

1. type関数

type 関数は、引数に渡されたオブジェクトのクラスを返す関数です。

print(type(someone))

>>> <class '\_\_main\_\_.Person'>

someoneはPersonクラスのインスタンスであることを証明しました。

print(type(Person))

>>> <class 'type'>

※　前の説明のように、Personクラスもメタクラスのオブジェクトです。typeはメタクラスのobjectの型です。<class 'type'>の説明はとても複雑なので、ここで省略させていただきます。ここでは、type関数の出力が<class 'type'>であることを満たすたびに、それがクラスであることを証明することを覚えておけば十分です。

1. isinstance関数

isinstance 関数は、引数を二つ受け取り、一つ目にオブジェクト、二つ目にクラスを受け取ります。一つ目に渡したオブジェクトが二つ目のクラスのインスタンスである場合はTrueを返します。

# someone **is an instance** of Person ?

# someoneがPersonのインスタンスですか

isinstance(someone, Person)

>>> True

* **上記のコメントのように英語でisinstance関数の作用が分かりやすいでしょう。**

help(isinstance)

>>> Help on built-in function isinstance in module builtins:

isinstance(obj, class\_or\_tuple, /)

Return whether an object is an instance of a class or of a subclass thereof.

A tuple, as in ``isinstance(x, (A, B, ...))``, may be given as the target to

check against. This is equivalent to ``isinstance(x, A) or isinstance(x, B)

or ...`` etc.

# **アクセス制御**

**6.2.1アトリビュート(attribute)の操作について**

class Person():

def \_\_init\_\_(self, name, age):

self.name = name

self.age = int(age)

def say(self):

print("私は", self.name, "と申します。", self.age, "歳です。", sep='')

位置パラメータで\_\_init\_\_関数の引数を定義する場合、必ず同じ数のパラメータ(selfパラメータ以外)を渡してインスタンスを初期化します。しないと、エラーが出てきます。

例：

someone = Person()

>>> ---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-5-1ac56b0c183e> in <module>

---> 1 someone = Person()

TypeError: \_\_init\_\_() missing 2 required positional arguments: 'name' and 'age'

上の例外の情報と同様に、2つの引数の初期値が欠けています。

キーワード引数で\_\_init\_\_関数の引数を定義する場合、引数の初期値がもうあるので、同じ数のパラメータ(selfパラメータ以外)を渡さなくても済みます。

class Person():

def \_\_init\_\_(self, name="無名", age=0):

self.name = name

self.age = int(age)

def say(self):

print("私は", self.name, "と申します。", self.age, "歳です。", sep='')

someone = Person()

someone.say()

>>> 私は無名と申します。0歳です。

また、あるインスタンスのすべてのアトリビュートの値はそのインスタンスの\_\_dict\_\_アトリビュートにあります。ここで、ちょっと不思議でしょう。一度も定義しなかった\_\_dict\_\_アトリビュートは一体なんでしょう。詳しくは継承（Inheritance）の節を参考してください。

print(someone.\_\_dict\_\_)

>>> {'name': '無名', 'age': 0}

従って、あるインスタンスの値を直接取得または変更できます。

print(someone.name)

print(someone.age)

>>> 無名

0

someone.name = "山田"

someone.age = 23

someone.say()

>>> 私は山田と申します。23歳です。

さらに、インスタンスに新しいアトリビュートを追加することもできます。新しいアトリビュートを追加する前に、まずアトリビュートの有無をチェックしましょう。

**6.2.2アトリビュートの有無チェック**

Pythonでは、指定のオブジェクトが特定の属性を持っているかを確認するhasattr関数があります。またここではgetattr関数についても取り扱います。

help(hasattr)

>>> Help on built-in function hasattr in module builtins:

hasattr(obj, name, /)

Return whether the object has an attribute with the given name.

This is done by calling getattr(obj, name) and catching AttributeError.

hasattr関数は、引数を二つ受け取り、一つ目にオブジェクト、二つ目にチェックしたいアトリビュートの名前を受け取ります。戻り値はTrueあるいはFalseです。

例：

print(hasattr(someone, 'name'))

>>> True

print(hasattr(someone, 'sex'))

>>> False

hasattr関数はgetattr関数を呼び出し、AttributeError例外型をキャッチすることによって行われます。getattr関数はオブジェクトから名前付きアトリビュートの値を取得するための関数です。getattr関数を使用した場合、保持していないアトリビュートを取得しようとするとAttributeErrorが発生しますが、第三引数でデフォルト値が設定されている場合はその値が返ります。

help(getattr)

>>> Help on built-in function getattr in module builtins:

getattr(...)

getattr(object, name[, default]) -> value

Get a named attribute from an object; getattr(x, 'y') is equivalent to x.y.

When a default argument is given, it is returned when the attribute doesn't

exist; without it, an exception is raised in that case.

例：

print(getattr(someone, 'name')) # 即ちsomeone.name

>>> 山田

print(getattr(someone, 'sex'))

>>> ---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-27-e3b679b299e1> in <module>

---> 1 print(getattr(someone, 'sex'))

AttributeError: 'Person' object has no attribute 'sex'

print(getattr(someone, 'sex', 'ない'))

>>> ない

では、インスタンスに新しいアトリビュートを追加しましょう。

class Person():

def \_\_init\_\_(self, name="無名", age=0):

self.name = name

self.age = int(age)

def say(self):

print("私は", self.name, "と申します。", self.age, "歳です。", sep='')

yamada = Person("山田", 40)

kuroiwa = Person("黒岩", 23)

suenaga = Person("末永", 33)

yamada.say()

kuroiwa.say()

suenaga.say()

>>> 私は山田と申します。40歳です。

私は黒岩と申します。23歳です。

私は末永と申します。33歳です。

print(hasattr(yamada, 'name'))

>>> True

print(hasattr(yamada, 'sex'))

>>> False

yamada.sex = "man"

print(yamada.sex)

>>> man

print(yamada.\_\_dict\_\_)

>>> {'name': '山田', 'age': 40, 'sex': 'man'}

このように、yamadaというインスタンスにsexというアトリビュートを追加しました。しかし、ほかのインスタンスにはこのsexというアトリビュートはありません。

print(hasattr(kuroiwa, 'sex'))

>>> False

print(kuroiwa.\_\_dict\_\_)

>>> {'name': '黒岩', 'age': 23}

**6.2.3インスタンスアトリビュート(Instance Attributes)とクラスアトリビュート(Class Attributes)**

インスタンスアトリビュート(Instance Attributes)とはそのインスタンスだけで使うアトリビュートです。クラスアトリビュート(Class Attributes)とはすべてのインスタンスで共有して使うアトリビュートで、すなわちそのクラスで使うアトリビュートです。クラスアトリビュートの値を変えれば、このクラスアトリビュートを共有しているすべてのインスタンスにこのアトリビュートが変わります。具体例を見ていきましょう。

例：

class Circle():

# 円周率の近似値をクラスアトリビュートとして定義

Pi = 3.1415926

# 円の半径を初期化

def \_\_init\_\_(self, radius):

self.radius = radius

# 円の周長を計算するメソッド

def perimeter(self):

return 2 \* self.Pi \* self.radius

# 円の面積を計算するメソッド

def area(self):

return self.Pi \* self.radius\*\*2

クラスアトリビュートはselfを使わずにメソッドの外に定義します。通常、インスタンスアトリビュートは\_\_init\_\_メソッドの中に、selfを使って定義します。

circle1 = Circle(3)

circle2 = Circle(5)

# クラスアトリビュートを出力

print(circle1.Pi)

print(circle2.Pi)

# すべてのインスタンスアトリビュートを出力

print(circle1.\_\_dict\_\_)

print(circle2.\_\_dict\_\_)

>>> 3.1415926

3.1415926

{'radius': 3}

{'radius': 5}

この例では円周率Piはクラスアトリビュートとして、オブジェクトcircle1とオブジェクトcircle2に共有されます。

perimeter1 = circle1.perimeter()

perimeter2 = circle2.perimeter()

print(perimeter1)

print(perimeter2)

>>> 18.849555600000002

31.415926

# クラスCircleのPi

Circle.Pi = 3.14

# クラスアトリビュートを出力

print(circle1.Pi)

print(circle2.Pi)

# すべてのインスタンスアトリビュートを出力

print(circle1.\_\_dict\_\_)

print(circle2.\_\_dict\_\_)

>>> 3.14

3.14

{'radius': 3}

{'radius': 5}

クラスCircleの円周率の値を変えれば、オブジェクトcircle1とオブジェクトcircle2の円周率の値も変わります。

# オブジェクトcircle1のPi

circle1.Pi = 3

print(circle1.Pi)

# クラスアトリビュートを出力

print(circle2.Pi)

# すべてのインスタンスアトリビュートを出力

print(circle1.\_\_dict\_\_)

print(circle2.\_\_dict\_\_)

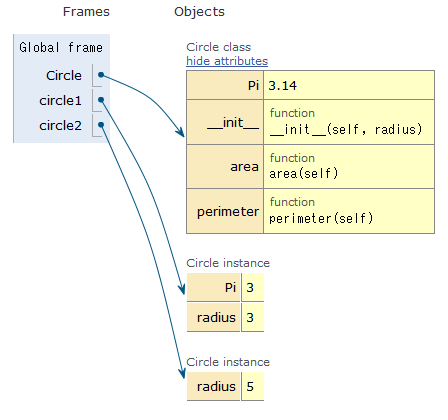
>>> 3

3.14

{'radius': 3, 'Pi': 3}

{'radius': 5}

オブジェクトcircle1のアトリビュートPiを変えると、オブジェクトcircle2のクラスアトリビュートPiも変わると思う人もいるでしょう。では、なぜここのオブジェクトcircle2のクラスアトリビュートPiが変わりませんのでしょうか。



実は、ここのcircle1.Pi = 3はクラスCircleの円周率の値を変えません。Pythonのクラスとクラスのインスタンスはそれぞれ、別々の区別された名前空間(namespace)[[8]](#footnote-8)をもっています。ただし、注意すべき点は、インスタンスの名前空間のほうがクラスの名前空間よりも優先されるということです。すなわち両方に同じ名前のアトリビュートが存在した場合, はじめにインスタンスの名前空間から適合するものをチェックし存在したらインスタンスのものを返却します。存在しなければクラスのものを探しに行くという手順を踏みます。従って、ここのcircle1.Pi = 3はインスタンスの名前空間にPiという新しいアトリビュートを作りました。そのアトリビュートの値は3です。

**6.2.4非公開アトリビュートのための名前修飾(Name Mangling)**

前の例では、インスタンスのアトリビュートの値を直接にクラスの外に変更しました。実際には、すべてのアトリビュートの値をクラスの外部で直接修正することを望んでいません。たとえば、クラスの内部計算に使用される一時的なアトリビュートを定義することもあるし、場合によっては、非常に重要なアトリビュートがあり、クラス外で簡単に変更できないようにする必要もあります。この点のために、Pythonは、クラス定義の外からは見えないようにすべきアトリビュートの命名方法(と言っても完全に隠せないが)を持っています。先頭に二つのアンダースコア(\_\_)を付けるのです[[9]](#footnote-9)。この命名方法を用いて定義されたアトリビュートは非公開アトリビュート(隠しアトリビュートあるいは隠し属性)と言われています。下記の例で理解しましょう。

例：

class Person():

def \_\_init\_\_(self, name, gender, age):

self.name = name

self.\_gender = gender # 先頭に一つのアンダースコア

self.\_\_age = int(age) # 先頭に二つのアンダースコア

def get\_age(self):

return self.\_\_age

sayaka = Person('sayaka', 'girl', 23)

print(sayaka.gender)

>>>---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-8-b218b718e161> in <module>

---> 1 print(sayaka.gender)

AttributeError: 'Person' object has no attribute 'gender'

print(sayaka.\_gender)

>>> girl

**先頭に一つのアンダースコアがあるアトリビュートは直接にクラスの外で訪問できませんが、クラスの外部で呼び出された時、アトリビュート名の前に一つのアンダースコアを追加するとアクセスすることもできます。**

print(sayaka.age)

>>>---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-2-54a0a2317b00> in <module>

---> 1 print(sayaka.age)

AttributeError: 'Person' object has no attribute 'age'

print(sayaka.\_\_age)

>>>---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-3-b31d0228d80a> in <module>

----> 1 print(sayaka.\_\_age)

AttributeError: 'Person' object has no attribute '\_\_age'

**一方で、先頭に二つのアンダースコアがあるアトリビュートは、クラスの外部で呼び出された時、アトリビュート名の前に二つのアンダースコアを追加しても、アクセスすることもできません。しかし、これはあくまでも人の目をごまかす方法です。実は、\_\_ageは\_classname\_\_ageへとテキスト置換されるようになるだけです。**

print(sayaka.\_\_dict\_\_)

>>> {'name': 'sayaka', '\_gender': 'girl', '\_Person\_\_age': 23}

print(sayaka.\_Person\_\_age)

>>> 23

**※ ほかのプログラミング言語のようなオブジェクトの中からしかアクセス出来ない "プライベート(private)" インスタンスのアトリビュートは、Pythonにはありません[[10]](#footnote-10)。**

**上記のような名前修飾(Name Mangling)はほとんどのPythonコードが従っている慣習の一つです。アンダースコアで始まる名前は、 (関数であれ、メソッドであれ、データメンバであれ)非公開なものとして扱います。これらはよくPythonのライブラリに現れます。ユーザーがそれを操作する必要がないことを示します。また、サブクラス(subclass)の操作にもよく使われます**[[11]](#footnote-11)**。**

**6.2.5プロパティ(property)によるアトリビュートのコントロール**

Pythonにはクラスの外でもアトリビュートの値が変えられます。この特性は便利ですが、セキュリティ面には隠れた危険があります。下記の例を見てください。

例：

class PersonalCard():

def \_\_init\_\_(self, name, age):

self.name = name

self.age = int(age) # ここで入力された年齢をint型に変換しても万全ですか？

def isdrinking(self):

if self.age > 20:

print("乾杯！")

else:

print("だめ！飲めないわ！")

上記のクラスがある酒場への切符を定義すると仮定します。人が飲むことができるかどうか年齢によって判断されます。

A\_naughty\_boy = PersonalCard('NoName', 14)

この行のコードは、14歳の少年の電子切符を定義しました。

A\_naughty\_boy.isdrinking()

>>> だめ！飲めないわ！

予想通り、彼は20歳未満なので、お酒が飲めません。

A\_naughty\_boy.age = 200

A\_naughty\_boy.isdrinking()

>>> 乾杯！

しかし、彼は天才肌のハッカーなので、ひそかに酒場のシステムに侵入して自分の実際の年齢を変えた結果、飲めるようになりました。そして、変更後の年齢は、世界で最も長生きした人の年齢よりも大きかったです。

A\_naughty\_boy.age = 'b'

わんぱくな少年は今回自分の年齢情報をあるアルファベットに変えました。驚いたことに、彼は無事に成功しました。

A\_naughty\_boy.isdrinking()

>>> ---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-31-9eb78775fdff> in <module>

---> 1 A\_naughty\_boy.isdrinking()

<ipython-input-24-ef63f3396e7c> in isdrinking(self)

5

6 def isdrinking(self):

---> 7 if self.age > 20:

8 print("乾杯！")

9 else:

TypeError: '>' not supported between instances of 'str' and 'int'

そして、最後に、酒場のシステムはクラッシュしてしまいました。

上記の例の問題は、アトリビュートの取得と設定を効果的に制御できないことです。ここで、Pythonのプロパティ(property)を登場させてもらおう。

class PersonalCard():

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age # ここもageアトリビュートを設置するので、ageのセッターメソッドを呼び出す

@property # ageアトリビュートにアクセスするためのゲッター(getter)メソッド

def age(self):

return self.\_\_age

@age.setter # ageアトリビュートを設置するためのセッター(setter)メソッド

def age(self, new\_age):

if not isinstance(new\_age, int):

print('警告！整数を入力して下さい')

elif new\_age < 0 or new\_age > 146:

print('もう一度入力ください')

else:

self.\_\_age = new\_age

def isdrinking(self):

if self.\_\_age > 20:

print("乾杯！")

else:

print("だめ！飲めないわ！")

@propertyと@アトリビュート名.setterの部分はPythonのデコレータ(decorator)と呼ばれています。その名の通り、デコレートは装飾(そうしょく)という意味です。デコレートを利用して、装飾を施された関数またはメソッドを統一的に処理できます。とても便利の機能です。Pythonにはたくさんの組み込みデコレータがあります。前の説明のように、Pythonの関数もオブジェクトです。実は、デコレータはだたの引数が関数(オブジェクト)である普通の関数です。その実現の原理も理解しやすいです。ここで@propertyと@アトリビュート名.setterの二つデコレータだけを紹介するので、興味がある方は自分で調べて学んでください。Pythonのデコレータは決して難しい概念ではありません。

上記のコードには二つのageというメソッドを定義しました。そして、@propertyと@アトリビュート名.setterで装飾します。装飾の目的はこの二つの関数はどちらがageアトリビュートにアクセスするためのゲッター(getter)メソッドであるか、どちらがageアトリビュートを設置するためのセッター(setter)メソッドであるかをPythonのインタプリタ(Interpreter)に教えるだけです。これで、ageアトリビュートを引き出すときにかならずageアトリビュートのゲッターメソッドを呼び出します。ageアトリビュートを設置するときにかならずageアトリビュートのセッターメソッドを呼び出します。

A\_naughty\_boy = PersonalCard('NoName', 'b')

>>> 警告！整数を入力して下さい

二番目の引数の型は整数ではないので、インスタンス化に失敗しました。

A\_naughty\_boy = PersonalCard('NoName', 14)

A\_naughty\_boy.age = 200

print(A\_naughty\_boy.age)

A\_naughty\_boy.isdrinking()

>>> もう一度入力ください

14

だめ！飲めないわ！

A\_naughty\_boy.age = 'b'

print(A\_naughty\_boy.age)

A\_naughty\_boy.isdrinking()

>>> 警告！整数を入力して下さい

14

だめ！飲めないわ！

今回、クラスの外で年齢を設置するとき、設定条件が満たされなかったため、設定に失敗しました。プロパティを用いて少なくともクラスのアトリビュートの操作を制限できます。実は、上記のPersonalCardクラスに本当の年齢に関するアトリビュートはageではなく、\_\_ageです。もちろん、この前の少年はまだ飲めるようになる手段がありますが、それは別の話です。プロパティをはじめ、これからもPythonのさまざまなデコレータを活用してください。それこそがPythonの便利さの一面です。

**6.2.6 \_\_slots\_\_によるアトリビュート名のコントロール**

この前のセクションで紹介したのように、Pythonでは誰でも自由にインスタンスに新しいアトリビュートを追加することができます。これもPythonの便利さの一面ですが、アトリビュートの名前と数を制限できるように、\_\_slots\_\_というクラスアトリビュートを使うこともよくあります。クラスのすべての利用できるアトリビュート名は\_\_slots\_\_に格納されます。一方で、Pythonではデフォルトでインスタンスのアトリビュートは\_\_dict\_\_を使って保存します。この保存の仕方では、実行中に新たなアトリビュートを動的に設定できたりして良いです。しかし、少数の固定なアトリビュートを持つ小さなクラスを扱うときには\_\_dict\_\_はメモリの無駄です。こういう時は\_\_slots\_\_にアトリビュートの名前を記述することでメモリを節約したほうが良いです。\_\_slots\_\_はよくさまざまなライブラリに運用されて、アトリビュート名を管理する役割を果たします。\_\_slots\_\_を使うと、クラスのすべてのアトリビュートも明瞭になります。

具体的な例を見ましょう。

class Person():

\_\_slots\_\_ = ['name', 'age', 'gender', 'height', 'weight']

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

someone = Person('sato', 40)

print(someone.name, someone.age)

>>> sato 40

print(hasattr(someone, 'gender'))

>>> False

someone.gender = 'man'

print(someone.gender)

>>> man

インスタンスに限定されたアトリビュートを追加することができます。

print(hasattr(someone, 'interest'))

>>> False

someone.interest = 'football'

>>> ---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-24-7a2762102d66> in <module>

---> 1 someone.interest = 'football'

AttributeError: 'Person' object has no attribute 'interest'

一方で、インスタンスに限定された内容以外のアトリビュートを追加することができません。

# **（Inheritance）**

**6.3.1一つのクラスの継承**

この前、「ヒト」というクラスを例として、クラスの概念を説明しました。この節に引き続き「ヒト」クラスを利用して、クラスの継承という概念を説明します。クラスの継承は名前の通り、基底クラス(またはスーパークラスsuperclass)の機能を持った派生クラス(またはサブクラスsubclass)を作るということです。「ヒト」の例を見ていきましょう。「ヒト」というクラスはさまざまなサブクラスに分けられることができます。たとえば、職業で分けると、科学者、歌手、俳優などに分けることができます。下の図6.3.1サブクラス(subclass)に示すように。

クラスの継承を使用する利点は、‘ヒト’というクラス(スーパークラス)を既に定義している場合、‘科学者’というクラスをゼロから定義する必要はありません。なぜなら、継承という手法を使用して、ヒトクラスをスーパークラスにして、科学者クラスをサブクラスにするだけで、ヒトのクラスにさらに機能を追加することができます。

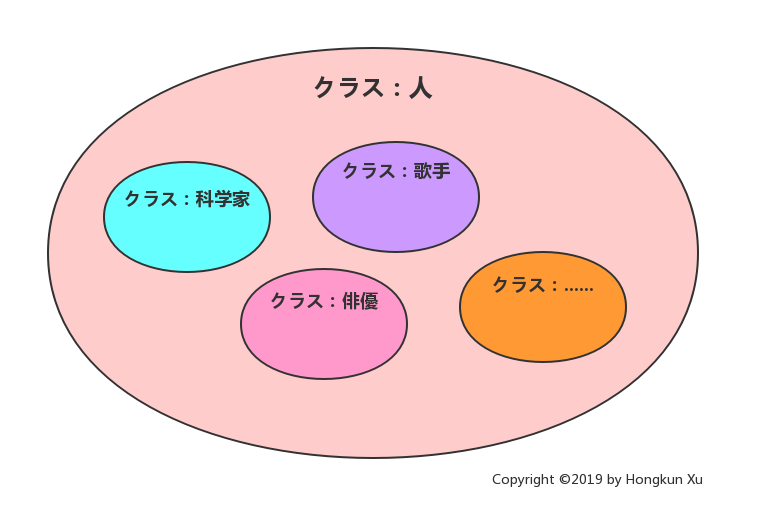


図6.3.1サブクラス(subclass)

この前のこの図[[12]](#footnote-12)はまだ覚えていますか。これはクラスの概念を説明するための図ですが、クラスの継承の説明のために、もう一度使わせていただきます。

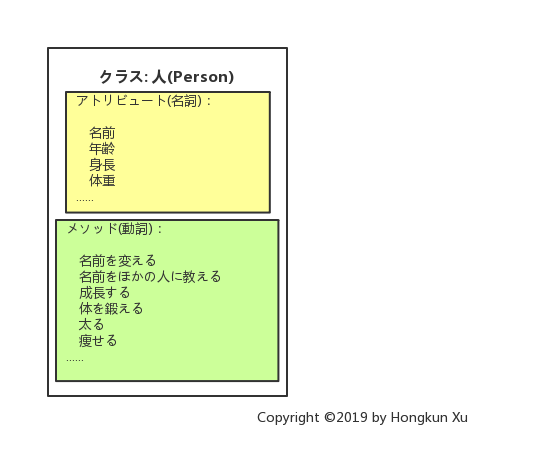


図6.3.2ヒトのクラス

ここで、上の「図6.3.1サブクラス(subclass)」の継承関係を再描画してみましょう。結果は「図6.3.3継承関係」の通りです。異なるサブクラスには、異なる新しいアトリビュートとメソッドがあります。科学者としては、提携機関で研究、または論文を書くことができます。歌手としては、自分のジャンルもあるし、当然ながら、歌うこともできます。そして、俳優としては、さまざまな芝居をすることにとって自分の演技力を高めることができます。

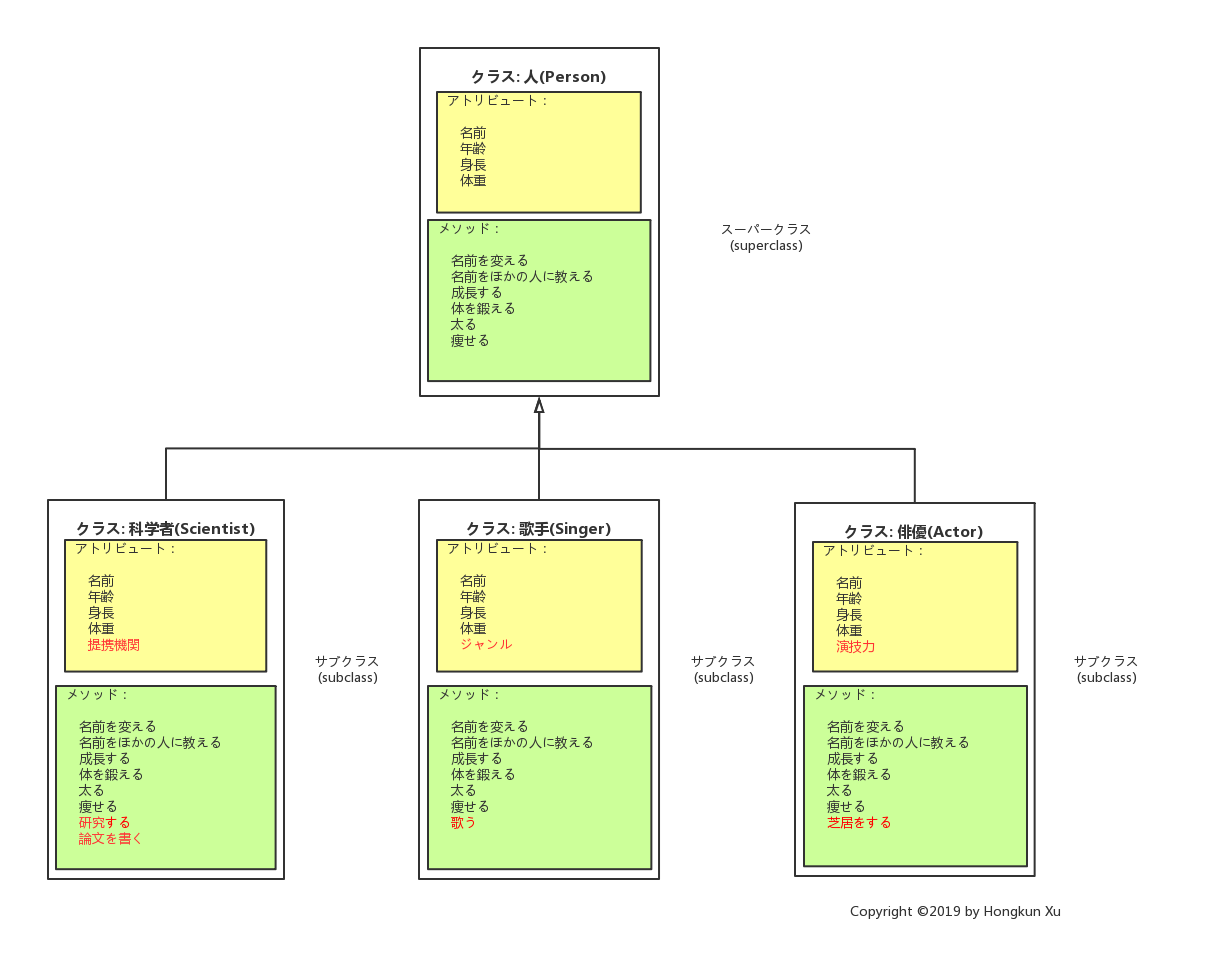


図6.3.3継承関係

上の図をさらに単純化すると(すべての単語を英語に書き直して、繰り返しの内容を省略する)、次の図のようになりました。

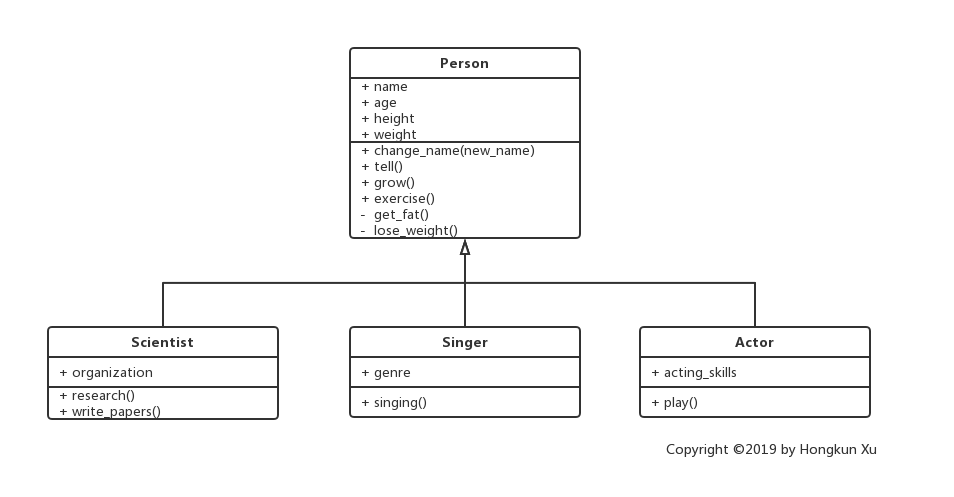


図6.3.4クラス図(Class diagram)

この図はクラス図(class diagram)[[13]](#footnote-13)と呼ばれています。クラス図は、システム開発でよく使用されるもので、クラス間の関係を明確に表現できます。これは、この章の最初のとこで使用された図たちよりもはるかに簡単ですね。+ 記号は公開のアトリビュートまたはメソッドを表し、- 記号は非公開のアトリビュートまたはメソッドを表します。これから内容に、クラス図のみを使用して説明させていただきます。

次に、Pythonでクラス図を参照しながら、実際のコードを書きましょう。

　クラスをどのようにPythonで定義するのかまだ覚えていますか。

class Person():

pass

Pythonでは、継承関係を表すのは非常に簡単です。下記のように、クラス名を宣言する時に、括弧の中に継承したいクラス名を入れることで継承ができます。

class Scientist(Person):

pass

class Singer(Person):

pass

class Actor(Person):

pass

クラス図を参照し、完全なPersonクラスは次のように定義されます。

class Person():

def \_\_init\_\_(self, name, age, height, weight):

self.name = name

self.age = age

self.height = height

self.weight = weight

def change\_name(self, new\_name):

if not isinstance(new\_name, str):

print('正しい名前を入力ください')

else:

self.name = new\_name

def tell(self):

print(f'私は{self.name}と申します。')

def grow(self):

self.height = self.height + 1

self.weight = self.weight + 0.1

def exercise(self):

if self.weight > 70:

self.\_lose\_weight()

elif self.weight < 50:

self.\_get\_fat()

def \_get\_fat(self):

self.weight = self.weight + 1

def \_lose\_weight(self):

self.weight = self.weight - 0.5

akira = Person('晶', 23, 65, 182)

akira.tell()

>>> 私は晶と申します。

akira.grow()

akira.change\_name('あきら')

akira.tell()

>>> 私はあきらと申します。

get\_fatとlose\_weightメソッドは、非公開のメソッドなので、先頭に\_を付けました。

サブクラスを初期化するとき、まずスーパークラスの\_\_init\_\_メソッドを呼び出すべきです。スーパークラスの\_\_init\_\_メソッドによって、継承されたアトリビュートは初期化することができます。このとき、super( )関数はよく使われます。

Python 2ではsuper( )関数の使い方はsuper(クラス, インスタンス自身).メソッド名( )です。一方で、Python 3ではsuper( )関数の引数を省略しても大丈夫です。super(クラスA, インスタンス自身).メソッドB( )の意味は、クラスAのインスタンスがクラスAのスーパークラスのメソッドBを呼び出します。super()関数は自動的にクラスAのスーパークラスを探します。したがって、直接にサブクラスのスーパークラス名を利用してスーパークラスのメソッドも呼び出せます。具体的な例を見てください。

例：

class Scientist(Person):

def \_\_init\_\_(self, name, age, height, weight, organization):

super(Scientist, self).\_\_init\_\_(name, age, height, weight)

self.organization = organization

def research(self):

pass

def write\_papers(self):

pass

例：

class Singer(Person):

def \_\_init\_\_(self, name, age, height, weight, genre):

# super括弧の中の内容を省略しても大丈夫です

super().\_\_init\_\_(name, age, height, weight)

self.genre = genre

def singing(self):

pass

例：

class Actor(Person):

def \_\_init\_\_(self, name, age, height, weight, acting\_skills):

# 直接にスーパークラス名を利用してスーパークラスのメソッドを呼び出す

Person.\_\_init\_\_(self, name, age, height, weight)

self.acting\_skills = acting\_skills

def play(self):

pass

この三つの方法のどれを使用しても、同じ効果を得ることができます。

Endo = Scientist('Endo',

84,

180,

70,

('Sankyo Co.', 'Tokyo University of Agriculture and Technology'))

print(Endo.name)

print(Endo.age)

print(Endo.organization)

>>> Endo

84

('Sankyo Co.', 'Tokyo University of Agriculture and Technology')

Endo.tell() # 継承したメソッド

>>> 私はEndoと申します。

koutaro = Singer('koutaro', 30, 185, 80, 'J-pop')

print(koutaro.\_\_dict\_\_)

>>> {'name': 'koutaro', 'age': 30, 'height': 185, 'weight': 80, 'genre': 'J-pop'}

koutaro.tell() # 継承したメソッド

>>> 私はkoutaroと申します。

suda = Actor('suda', 28, 175, 65, 90)

print(suda.acting\_skills)

>>> 90

suda.tell() # 継承したメソッド

>>> 私はsudaと申します。

* この前の\_\_dict\_\_アトリビュートの話[[14]](#footnote-14)はまだ覚えているでしょうが、なぜ一度も定義しなかった\_\_dict\_\_アトリビュートが使えますか。実は、これはPythonすべてのクラスのスーパークラスのobjectの‘魔法’です。

Print(object.\_\_dict\_\_)

>>> mappingproxy({'\_\_repr\_\_': <slot wrapper '\_\_repr\_\_' of 'object' objects>,

'\_\_hash\_\_': <slot wrapper '\_\_hash\_\_' of 'object' objects>,

'\_\_str\_\_': <slot wrapper '\_\_str\_\_' of 'object' objects>,

'\_\_getattribute\_\_': <slot wrapper '\_\_getattribute\_\_' of 'object' objects>,

'\_\_setattr\_\_': <slot wrapper '\_\_setattr\_\_' of 'object' objects>,

'\_\_delattr\_\_': <slot wrapper '\_\_delattr\_\_' of 'object' objects>,

'\_\_lt\_\_': <slot wrapper '\_\_lt\_\_' of 'object' objects>,

'\_\_le\_\_': <slot wrapper '\_\_le\_\_' of 'object' objects>,

'\_\_eq\_\_': <slot wrapper '\_\_eq\_\_' of 'object' objects>,

'\_\_ne\_\_': <slot wrapper '\_\_ne\_\_' of 'object' objects>,

'\_\_gt\_\_': <slot wrapper '\_\_gt\_\_' of 'object' objects>,

'\_\_ge\_\_': <slot wrapper '\_\_ge\_\_' of 'object' objects>,

'\_\_init\_\_': <slot wrapper '\_\_init\_\_' of 'object' objects>,

'\_\_new\_\_': <function object.\_\_new\_\_(\*args, \*\*kwargs)>,

'\_\_reduce\_ex\_\_': <method '\_\_reduce\_ex\_\_' of 'object' objects>,

'\_\_reduce\_\_': <method '\_\_reduce\_\_' of 'object' objects>,

'\_\_subclasshook\_\_': <method '\_\_subclasshook\_\_' of 'object' objects>,

'\_\_init\_subclass\_\_': <method '\_\_init\_subclass\_\_' of 'object' objects>,

'\_\_format\_\_': <method '\_\_format\_\_' of 'object' objects>,

'\_\_sizeof\_\_': <method '\_\_sizeof\_\_' of 'object' objects>,

'\_\_dir\_\_': <method '\_\_dir\_\_' of 'object' objects>,

'\_\_class\_\_': <attribute '\_\_class\_\_' of 'object' objects>,

'\_\_doc\_\_': 'The most base type'})

上記の出力結果はすべてのPythonクラスの基本的なアトリビュートです。

**6.3.2複数クラス継承**

　Pythonは多重継承(Multiple inheritance)をサポートする言語の一つです。複数クラス継承の方法も容易いです。下記のように、クラス名を宣言する時に、括弧の中に継承したい複数のクラス名を入れることで継承ができます。

class Star(Singer, Actor):

pass

　しかし、複数クラス継承によって、さまざまなことにも注意しなければなりません。たとえば、サブクラスのインスタンスを初期化するとき、どの順番で複数のスーパークラスの引数を初期化するのは良いですか。または、上記の例のように、継承された複数のスーパークラスは同じスーパークラス(同じスーパークラスの同じ引数の初期化問題)がある場合、どのような方法で、同じ引数を初期化するのかなどいろいろな注意点があります。では、ここで、上記の難題を解決しましょう。

理解しやすいために、これらのクラスの定義を簡略化させていただきます。super()関数を用いる場合は、以下のようになります。

class Person():

def \_\_init\_\_(self, name):

self.name = name

class Singer(Person):

def \_\_init\_\_(self, name, genre):

super().\_\_init\_\_(name)

self.genre = genre

class Actor(Person):

def \_\_init\_\_(self, name, acting\_skills):

super().\_\_init\_\_(name)

self.acting\_skills = acting\_skills

class Star(Singer, Actor):

def \_\_init\_\_(self, name, genre, acting\_skills):

super().\_\_init\_\_(name, genre, acting\_skills) # errorが出る

starA = Star('suda', 'J-pop', 90)

>>> ---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-40-a0e43352c06d> in <module>

----> 1 starA = Star('suda', 'J-pop', 90)

<ipython-input-39-363270b02ef0> in \_\_init\_\_(self, name, genre, acting\_skills)

18 class Star(Singer, Actor):

19 def \_\_init\_\_(self, name, genre, acting\_skills):

---> 20 super().\_\_init\_\_(name, genre, acting\_skills)

21 print('{}クラスの初期化を行った'.format(self.\_\_class\_\_.\_\_name\_\_))

22

TypeError: \_\_init\_\_() takes 3 positional arguments but 4 were given

想像した通り、何もせずにStarクラスのインスタンスを初期化すると、エラーが出てきます。例外の情報によると、定義されたパラメータの数は実際に代入する引数の数よりすくないです。実は、あるクラスと他のクラスが一緒に継承される時、他のクラスの引数を知らないといけないなんて、直感的におかしいと分かるでしょう。したがって、このようなことを避けるため、多重継承する時、\_\_init\_\_を一個に限定するというミックスイン(Mix-in)原則が提出されました。つまり、継承されるクラス群の中に、最大一個だけ\_\_init\_\_を持つことが許されます。この原則にしたがって、もう一度修正した結果はこちらです。

class Person():

def \_\_init\_\_(self, name):

self.name = name

print('Personクラス')

class Singer(Person):

# Starクラスの\_\_init\_\_()と同じ数の引数になる

def \_\_init\_\_(self, name, genre, acting\_skills):

# Starクラスの\_\_init\_\_()と同じ数の引数になる

super().\_\_init\_\_(name, genre, acting\_skills)

self.genre = genre

print('Singerクラス')

class Actor(Person):

# Starクラスの\_\_init\_\_()と同じ数の引数になる

def \_\_init\_\_(self, name, genre, acting\_skills):

super().\_\_init\_\_(name)

self.acting\_skills = acting\_skills

print('Actorクラス')

class Star(Singer, Actor):

def \_\_init\_\_(self, name, genre, acting\_skills):

super().\_\_init\_\_(name, genre, acting\_skills)

print('Starクラス')

starA = Star('suda', 'J-pop', 90)

>>> Personクラス

Actorクラス

Singerクラス

Starクラス

今回、無事に実行しました。出力情報によると、実行の順番は下図のようになりました。

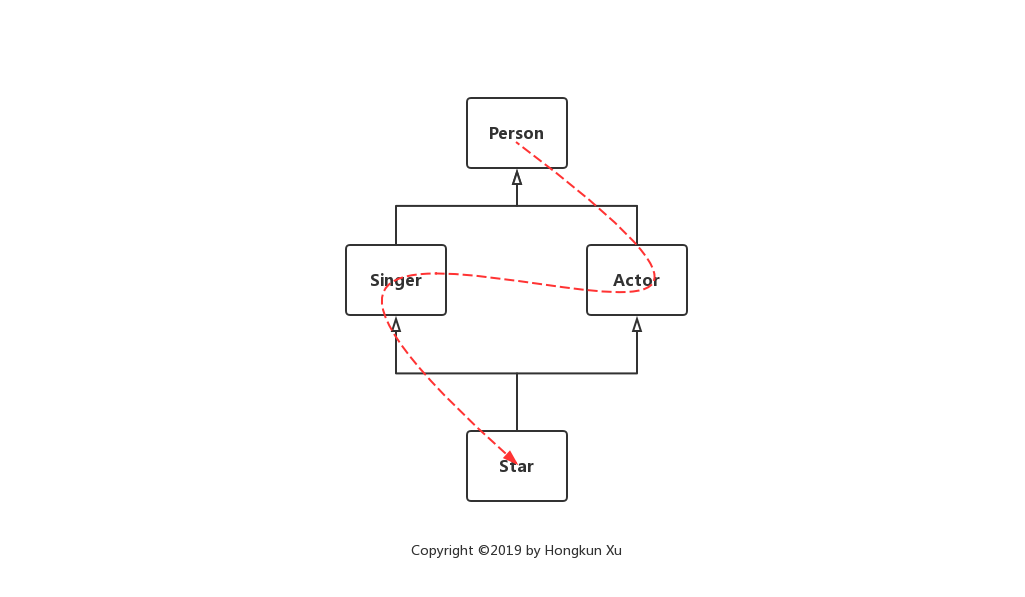


図6.3.5多重継承(Multiple inheritance)

では、Starクラスの実の継承順番のリストを見ましょう[[15]](#footnote-15)。

Star.\_\_mro\_\_

>>> (\_\_main\_\_.Star, \_\_main\_\_.Singer, \_\_main\_\_.Actor, \_\_main\_\_.Person, object)

この結果の理由は、Pythonがデフォルトでクラスが定義されているときに、括弧の中で継承されたクラスの順序によってインスタンスを初期化することです。Pythonのデフォルトの多重継承の順序がまだ理解できない場合、下記のコード通り、\_\_init\_\_メソッドの引数を変えずスーパークラス名を用いてクラスを定義したほうがいいです。

class Person():

def \_\_init\_\_(self, name):

self.name = name

print('Personクラス')

class Singer(Person):

def \_\_init\_\_(self, name, genre):

Person.\_\_init\_\_(self, name) # スーパークラス名を用いて定義する

self.genre = genre

print('Singerクラス')

class Actor(Person):

def \_\_init\_\_(self, name, acting\_skills):

Person.\_\_init\_\_(self, name) # スーパークラス名を用いて定義する

self.acting\_skills = acting\_skills

print('Actorクラス')

class Star(Singer, Actor):

def \_\_init\_\_(self, name, genre, acting\_skills):

Actor.\_\_init\_\_(self, name, acting\_skills) # スーパークラス名を用いて定義する

Singer.\_\_init\_\_(self, name, genre) # スーパークラス名を用いて定義する

print('Starクラス')

starA = Star('suda', 'J-pop', 90)

>>> Personクラス

Actorクラス

Personクラス

Singerクラス

Starクラス

二重初期化によって、Personクラスの\_\_init\_\_メソッドは2回呼び出されました。スーパークラス名を用いて定義されたクラスの継承関係は明瞭で理解しやすいです。下記のコードのように、スーパークラス名で継承したクラスの呼び出す順番を変えると、初期化順番も異なります。

class Star(Singer, Actor):

def \_\_init\_\_(self, name, genre, acting\_skills):

Singer.\_\_init\_\_(self, name, genre) # スーパークラス名を用いて定義する

Actor.\_\_init\_\_(self, name, acting\_skills) # スーパークラス名を用いて定義する

print('Starクラス')

starA = Star('suda', 'J-pop', 90)

>>> Personクラス

Singerクラス

Personクラス

Actorクラス

Starクラス

単一継承は、多重継承よりも広く使用されています。多重継承にまだなれない方は、単一継承を使用することをお勧めします。クラス継承の概念や方法はオブジェクト指向プログラミングにおいては、頻繁に使うものですので、ぜひ使いこなしていきましょう。

**豆知識：標準的なクラスの定義方法**

これまでのところ、Pythonクラスに関するさまざまな知識を紹介いたしました。こちらの知識を用いて実際のコードを書く時、プログラミングツールが必要になります。プログラミングツールの多くには、コード補完機能が付いています。たとえば、「class」を入力すると、自動的に標準的なクラスの定義書式が生成できます。下記のコードはSublime textというツールの生成結果です。これは、ほぼ完璧なクラスの定義方法です。Pythonのクラスの特性を十分理解する上で、このようなクラスの定義方法を使いましょう。

**class ClassName(object):**

**"""docstring for ClassName"""**

**def \_\_init\_\_(self, arg):**

**super(ClassName, self).\_\_init\_\_()**

**self.arg = arg**

# **ポリモーフィズム（Polymorphism）**

この節は継承の続きとなります。前回であらかた継承の話はしたのですが、これからはその発展となります。ポリモーフィズム(Polymorphism)とは、日本語でと言います。多態性とは、クラスの種類(変数の型)に関わらず、異なるオブジェクトに対して同じ操作を適用することです。説明を聞いてもわかりにくいと思うので、具体的な例を見ていきましょう。

class Person():

def run(self):

print('A person is runing.')

class Teacher(Person):

def run(self):

print('A teacher is runing.')

class Animal():

def run(self):

print('An animal is runing.')

a = Person()

b = Teacher()

c = Animal()

def tell(ins):

ins.run()

tell(a)

tell(b)

tell(c)

>>> A person is runing.

A teacher is runing.

An animal is runing.

上記のa,b,cは異なるクラスのオブジェクトですが、同名のメソッドがあります。特に注意しなければならないのは、TeacherクラスはPersonクラスのサブクラスで、Personクラスと同じ名前のメソッドがあります。多態性によって、同名のメソッドまたは関数で異なる処理ができます。これはごく普通だと思う方もいらっしゃるでしょう。実は、ポリモーフィズムは大きな役割を果たしました。下記の例を見てください。

a = [1,2,3,4] # リストのインスタンス

b = 'tree' # 文字列のインスタンス

c = (a, b) # タプルのインスタンス

print(len(a))

print(len(b))

print(len(c))

>>> 4

4

2

ここのlen()関数は異なるオブジェクトを統一的な処理を行うことができるのはポリモーフィズムのおかげです。ポリモーフィズムがなければ、異なるメソッドに異なる名前をつけなければなりません。そして、すべての異なる名前を覚えておく必要があります。これは想像するだに恐ろしいことです。実は、我々は無意識のうちに、ポリモーフィズムの利点を利用してきました。

C++などの言語の経験がある方は、きっとC++で異なる型の変数あるいはアトリビュートを同じ操作にするのはどのようなややこしい処理が必要なのかをわかるでしょう。Pythonの場合、そちらのややこしい処理はもうすでに自動的に完成しました。

1. **基礎になったライブラリ**

# **Pythonのライブラリ**

「、其の事を善くせんと欲すれば、必ず先ず其の器をくす。」

――孔子『論語』

# **標準ライブラリ**

### **math**

### **collection**

### **datetime**

### **decimal**

# **よく利用されるライブラリ**

### **numpyによるテータ処理**

**つづく**

**参考資料**

Python 3.6.5 ドキュメント(日本語版)　<https://docs.python.jp/3/index.html>

Bill Lubanovic（2017）『入門 Python 3（Introducing Python）』（斎藤 康毅監修, 長尾 高弘翻訳）オライリー

Luciano Ramalho（2015）『Fluent Python』(中国語版) O’Reilly

David Beazley, Brian K. Jones（2013）『Python Cookbook（Third Edition）』（原版）O’Reilly

****

**潜竜、用うる勿れ**

**見龍田に在り**

**終日乾乾す**

**あるいは躍りて淵に在り**

**飛龍天に在り**

**亢龍悔あり**

1. ハンドラ：ここには、一つのexcept節中の処理内容という意味です。 [↑](#footnote-ref-1)
2. 注：変換型は変換指定子のパーセント%の後ろの部分です [↑](#footnote-ref-2)
3. 後の節で説明します [↑](#footnote-ref-3)
4. コンストラクタ(Constructor)とは、オブジェクトが生成される時に実行されるメソッドのことです。オブジェクト(インスタンス)が扱う変数などの初期化を主に行います。 [↑](#footnote-ref-4)
5. Python 3.7.2 公式ドキュメントより [↑](#footnote-ref-5)
6. Pythonのガベージコレクションメカニズム(Garbage Collection Mechanism)は、参照カウントだけではなく、マーク・アンド・スイープ(mark-and-sweep)方法と世代別ガベージコレクション(Generational Garbage Collection)方法も含む優れたガベージコレクションメカニズムです。 [↑](#footnote-ref-6)
7. Python 3.7.2 公式ドキュメントより [↑](#footnote-ref-7)
8. 名前空間(namespace)とは、変数や関数の名前が所属している場所のことです。変数や関数が所属する場所、つまりクラスやモジュール(次に章で説明する)が名前空間になります。異なる名前空間にある同じ名前の変数や関数には何の関係もないということを覚えておきましょう。例えば、A, Bという二つのクラスがあるとき、それぞれに同じ名前のアトリビュートやメソッドを定義することができます。 [↑](#footnote-ref-8)
9. 末尾に一個以下のアンダースコア [↑](#footnote-ref-9)
10. Python 3.6.5 公式ドキュメントより [↑](#footnote-ref-10)
11. 詳しくは継承の節を参考してください [↑](#footnote-ref-11)
12. 6.1.2オブジェクト指向プログラミング(Object Oriented Programming)節の図 [↑](#footnote-ref-12)
13. クラス図とは、クラス、属性、クラス間の関係からシステムの構造を記述する静的な構造図である。 [↑](#footnote-ref-13)
14. 6.2.1アトリビュート(attribute)の操作についての節の問題 [↑](#footnote-ref-14)
15. \_\_mro\_\_はクラスのアトリビュートとして、受け取ったクラスの祖先クラスを「名前解決を行う順番に並べた上で」リストに格納してタプルを返してくれます。 [↑](#footnote-ref-15)