Python3 チートシート[*1]

目次

[*1] https://qiita.com/1429takahiro/items/710a877b1afb1626334f を元に組版

1 Python の基本

1.1 基本文法

インデント

```
import␣os
def␣function():
␣␣␣␣#␣インデントは半角スペース 4つが推奨されている
␣␣␣␣print('Hello␣world')
```

変数宣言

```
# 数値
num = 1
# 文字列
name = 'Tanaka'
# リスト
list = [1, 2, 3, 4, 5]

# 明示的な型宣言
num: int = 1
name: str = 'Tanaka'

# 型変換
old_num = '1' # String 型
new_num = int(num) # integer 型に変換して new_num に代入
```

- Python で変数宣言が不要な理由は、Python が動的型付き言語の為。動的型付き言語はプログラムの実行中に型を自動で判定しながら処理を行っていく。型を明示しなくても実行時には決まっているが、暗黙的な型変換は行われない為、コード記述時には型を意識する必要がある。

コメント

```
# 1行単位のコメント

"""
複数行を
まとめて
コメント
"""
```

Docstring

```
def test_function(param1, param2):
    """二つの引数を加算して返却するtest_function

    Args:
        param1(int): 第一引数の説明
        param2(int): 第二引数の説明

    Returns:
        param1 と param2 の加算結果を返却する
    """
    return param1 + param2
```

- Docstring はコメントと違い様々なツールからも参照できるドキュメント。モジュールの冒頭と関数、クラス、メソッドの定義のみで利用可能。(関数等の途中では使えない。)

1.2 数値の操作

数値の操作

```
# 整数の加算
>>> 1 + 1
2

# 整数の減算
>>> 2 - 1
1

# 整数の乗算
>>> 5 * 6
30

# 整数の除算 ※除算は常に浮動小数点を返す
>>> 6 / 5
1.2
```

```
# 評価順は数学と同様
>>> 50 - 5 * 6
20
>>> (50 - 5 * 6) / 4
5.0

# 標準の除算はfloat を返す
>>> 17 / 3
5.666666666666667

#「//」:切り下げ除算は小数点以下を切り捨てる
>>> 17 // 3
5

#「%」:剰余(除算の余り)を返す
>>> 17 % 3
2
>>>

# 5の 2乗
>>> 5 ** 2
25
# 2の 7乗
>>> 2 ** 7
128

# 小数点以下を丸める(小数点以下を 2桁で丸める)
>>> pie = 3.1415926535
>>> pie
3.1415926535
>>> round(pie, 2)
3.14
```

1.3 文字列の操作

文字列

```
# シングルクオートで囲む
>>> print('Hello world')
Hello world

# ダブルクオートで囲む
>>> print("Hello world")
Hello world
```

```
# 文字列として「’」を使う場合（ダブルクオートで囲む）
>>> print("I’m from Japan.")
I’m from Japan.

# 文字列として「’」を使う場合（「\」—日本語キーボードでは小文字の「￥」—でエスケープする）
>>> print('I\'m from Japan.')
I’m from Japan.

# 改行する（\n）
>>> print('Hello!\nHow are you doing?')
Hello!
How are you doing?

# 特殊文字としての解釈を防ぐ（「\n」部分が改行として解釈される為、raw データとして処理させる）
>>> print(r'C:\name\name')
C:\name\name

# 複数行の出力（「"""」で囲む）
>>> print("""
... line1
... line2
... line3
... """)

line1
line2
line3

# 改行をさせない（「\」を記載する）
>>> print("""\
... line1
... line2
... line3\
... """)
line1
line2
line3

# 文字の繰り返し
>>> print('Yes.' * 3 + '!!!')
Yes.Yes.Yes.!!!

# 文字列の結合（リテラル同士は「+」がなくてもOK）
>>> print('Py''thon')
Python
```

```
# 文字列の結合（変数 + リテラルは「+」が必要）
>>> prefix = 'Py'
>>> print(prefix + 'thon')
Python
```

文字列のインデックス/スライシング

```
# 変数定義
>>> word = 'Python'

# word の 0 番目の位置の文字を取得
>>> print(word[0])
P

# word の 5 番目の位置の文字を取得
>>> print(word[5])
n

# word の最後から 1 番目の位置の文字を取得（先頭からは、0,1,2・・・　最後からは-1,-2,-3・・・）
>>> print(word[-1])
n

# word の最後から 3 番目の位置の文字を取得（先頭からは、0,1,2・・・　最後からは-1,-2,-3・・・）
>>> print(word[-3])
h

# word の 0（0含む）から 2（2は含まない）の位置まで文字を取得
>>> print(word[0:2])
Py

# word の 2（2含む）から 5（5は含まない）の位置まで文字を取得
>>> print(word[2:5])
tho

# word の最初から 2（2含む）の位置まで文字を取得
>>> print(word[:2])
Py

# word の 2（2含む）の位置から最後までの文字を取得
>>> print(word[2:])
thon

# word の最初から最後までの文字を取得
>>> print(word[:])
Python
```

文字に関するメソッド

```
# 文字列s を定義
>>> s = 'Hello Tom. How are you doing.'
>>> print(s)
Hello Tom. How are you doing.

# 文字列に特定の文字列が含まれるかを確認
>>> print(s.startswith('Hello'))
True
>>> print(s.startswith('Tom'))
False

# 文字列に特定の文字 (or 文字列)が含まれる位置を確認 (先頭から)
>>> print(s.find('o'))
4

# 文字列に特定の文字 (or 文字列)が含まれる位置を確認 (最後から)
>>> print(s.rfind('o'))
24

# 文字列に含まれる特定の文字 (or 文字列)の数を確認
>>> print(s.count('o'))
5

# 文字列の先頭の文字を大文字に変換
>>> print(s.capitalize())
Hello tom. how are you doing.

# 文字列の各単語の先頭の文字を大文字に変換
>>> print(s.title())
Hello Tom. How Are You Doing.

# 文字列を大文字に変換
>>> print(s.upper())
HELLO TOM. HOW ARE YOU DOING.

# 文字列を小文字に変換
>>> print(s.lower())
hello tom. how are you doing.

# 特定の文字列を置換
>>> print(s.replace('Tom', 'Bob'))
Hello Bob. How are you doing.
```

2 制御構造

2.1 条件分岐 (if 文)

条件分岐 (if 文)

```
if 条件式 1:
    条件式 1が真の場合に実行される処理
elif 条件式 2:
    条件式 1が偽かつ条件式 2が真の場合に実行される処理
else:
    全ての条件式が偽の場合に実行される処理
```

- if は上から順に評価され、最初に真になった節の処理が実行される。条件式 1 が真の場合、条件式 2 の結果が真でも条件式 2 の処理は実行されない。

条件式で偽となる値

```
- None
- False
- 数値型のゼロ 、0 、0.0 、0j(複素数)
- 文字列､リスト､辞書､集合などのコンテナオブジェクトの空オブジェクト
- メソッド__bool__()がFalse を返すオブジェクト
- メソッド__bool__()を定義しておらず､メソッド__len__()が 0を返すオブジェクト
```

- Python では、偽となるオブジェクト以外が真と評価される。つまり上記以外は全て真となる。

数値の比較

```
# 等価の場合にTrue
>>> 1 == 1
True

# 等価でない場合にTrue
>>> 1 != 1
False

# 左辺が大きい場合にTrue
>>> 1 > 0
True

# 右辺が大きい場合にTrue
>>> 1 < 0
False

# 左辺が大きいまたは等価の場合にTrue
```

```
>>> 1 >= 0
True

# 右辺が大きいまたは等価の場合にTrue
>>> 1 <= 0
False

# x < y and y < z と等価
>>> x, y, z = 1, 2, 3
>>> x < y < z
True
```

オブジェクトの比較

```
>>> x = 'red'
>>> y = 'green'

# 等価の場合にTrue
>>> x == y
False

# 等価でない場合にTrue
>>> x != y
True

# 同じオブジェクトの場合にTrue
>>> x is None
False

# 同じオブジェクトでない場合にTrue
>>> x is not None
True

>>> items = ['pen', 'note', 'book']

# items に'book'が含まれている場合にTrue
>>> 'book' in items
True

# items に'note'が含まれていない場合にTrue
>>> 'note' not in items
False
```

2.2 ループ（for 文）

for 文

```
>>> items = [1, 2, 3]
>>> for i in items:
...     print(f'変数i の値は{i}')
...
変数i の値は 1
変数i の値は 2
変数i の値は 3

# range()関数
>>> for i in range(3):
...     print(f'{i}番目の処理')
...
0番目の処理
1番目の処理
2番目の処理

# enumerate()関数
>>> chars = 'word'
>>> for count, char in enumerate(chars):
...     print(f'{count}番目の文字は{char}')
...
0番目の文字はw
1番目の文字はo
2番目の文字はr
3番目の文字はd

# ディクショナリのループ（items()メソッド）
>>> sports = {'baseball': 9, 'soccer': 11, 'tennis': 2}
>>> for k, v in sports.items():
...     print(k, v)
...
baseball 9
soccer 11
tennis 2

# 二つのシーケンスの同時ループ（zip()関数）
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue' ]
>>> for q, a in zip(questions, answers):
...     print('What is your {0}? It is {1}.'.format(q, a))
...
```

```
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

for 文の else 節の挙動

```
>>> nums = [2, 4, 6, 8]
>>> for n in nums:
...     if n % 2 == 1:
...         break
... else:
...     print('奇数がありません')
...
奇数がありません
```

- else 節には for 文が終了した後に一度だけ実行したい処理を記述する。
- else 節を使用すると、for 文、あるいは while 文の処理中で break 文を使用しなかった時、else 節のブロックを実行する。

2.3 例外処理

try 文

```
try:
    例外が発生する可能性のある処理
except 捕捉したい例外クラス:
    捕捉したい例外が発生した時に実行される処理
else:
    例外が発生しなかった時のみ実行される処理
finally:
    例外の発生有無にかかわらず実行したい処理
```

- try 節：例外が発生する可能性のある処理を try-except 節の間に記載する

try-except 節

```
# 配列の存在しない要素へのアクセスを行い例外を発生させ、except 節で捕捉する
>>> l = [1, 2, 3]
>>> i = 5
>>> try:
...     l[i]
... except:
...     print('It is no problem')
...
It is no problem
```

```
# 特定の例外をexcept 節で捕捉する
>>> try:
...     l[i]
... except IndexError:
...     print('You got an IndexError')
...
You got an IndexError

# 例外の原因を出力させる
>>> try:
...     l[i]
... except IndexError as ex:
...     print('You got an IndexError : {}'.format(ex))
...
You got an IndexError : list index out of range

# 全ての例外を捕捉する
>>> try:
...     () + l
... except IndexError:
...     print('You got an IndexError')
... except BaseException as ex:
...     print('You got an Exception : {}'.format(ex))
...
You got an Exception : can only concatenate tuple (not "list") to tuple
```

- else 節：例外が発生しなかった場合に実行する処理を else 節に記述する

else 節

```
# 例外が発生しなかった場合に実行する処理をelse 節に記述する
>>> try:
...     1 + 1
... except BaseException:
...     print('You got an Exception.')
... else:
...     print('There are no Exceptions!')
...
2
There are no Exceptions!
```

- finally 節：例外が発生しても発生しなくても実行する処理を finally 節に記述する

finally 節

```
# 例外が発生しても発生しなくても実行するfinally 節に記述する
>>> try:
...     raise NameError('It is a NameError!!!')
... except NameError as ne:
...     print(ne)
... finally:
...     print('NameError Program finished!!!')
...
It is a NameError!!!
NameError Program finished!!!
```

例外の送出（意図的な例外の発生）

raise 文

```
# 単純にraise のみを記載すると、例外を再送出することができる。
#(例外の送出は知りたいが、その場では処理をさせない場合)
>>> try:
...     raise NameError('It is a NameError!!!')
... except NameError:
...     print('Name Error.')
...     raise
...
Name Error.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: It is a NameError!!!
```

3 データ構造

3.1 リスト型

リスト型

```
# リスト「l」の定義
>>> l = [1, 2, 3, 4, 5]
>>> print(l)
[1, 2, 3, 4, 5]

# リストの特定の要素の取得
>>> print(l[0])
1
>>> print(l[-1])
5

# リストの最初から 3番目（0番目、1番目、2番目）の要素まで（3番目は含まない）を取得
```

```
>>> print(l[:3])
[1, 2, 3]

# リストの最後から 3番目（3番目は含む）の要素から最後までを取得
>>> print(l[-3:])
[3, 4, 5]

# リストの全ての要素を取得
>>> print(l[:])
[1, 2, 3, 4, 5]

# リストの結合
>>> l2 = [6, 7, 8, 9, 10]
>>> print(l + l2)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# リストの要素の変更
>>> l[2] = 300
>>> print(l)
[1, 2, 300, 4, 5]
```

- 文字列は変更不能体（immutable）であり、特定の要素を変更することはできないが、リストは変更可能体（mutable）であり、要素を入れ替えることができる。

3.2 リストの操作

リストの操作

```
# リストの要素の追加（最後に追加）
>>> l.append(600)
>>> print(l)
[1, 2, 300, 4, 5, 600]

# リストの要素の追加（先頭に追加：0番目のインデックスに「0」を追加）
>>> l.insert(0, 0)
>>> print(l)
[0, 1, 2, 300, 4, 5, 600]

# リストの要素の取得（最後の要素の取得、リストからは消える）
>>> l.pop()
600
>>> print(l)
[0, 1, 2, 300, 4, 5]

# リストの要素の取得（０番目の要素の削除、リストからは消える）
```

```
>>> l.pop(0)
0
>>> print(l)
[1, 2, 300, 4, 5]

# リストの要素の削除（2番目の要素の削除）
>>> del l[2]
>>> print(l)
[1, 2, 4, 5]

# リストの削除(リストごと削除)
>>> del l
>>> print(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'l' is not defined

# リストの全要素の削除(リストは削除しない)
>>> print(l)
[['A', 'B', 'C'], [1, 2, 3]]
>>> l.clear()
>>> print(l)
[]

# リストの指定した値に合致した要素を削除
>>> l = [0, 1, 2, 3, 3, 3, 4, 5]
>>> l.remove(3)
>>> print(l)
[0, 1, 2, 3, 3, 4, 5]

# リストの指定した値に合致した要素を削除(合致する要素がないとエラーとなる)
>>> l.remove(3)
>>> l.remove(3)
>>> l.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> print(l)
[0, 1, 2, 4, 5]

# リストの要素数の取得
>>> print(len(l))
5

# リストの入れ子(リストを要素とするリストを作成)
>>> l1 = ['A', 'B', 'C']
```

```
>>> l2 = [1, 2, 3]
>>> l = [l1, l2]
>>> print(l)
[['A', 'B', 'C'], [1, 2, 3]]

# 指定した値に合致するインデックスの取得(アイテムが存在しない場合はエラー)
>>> l = [1, 2, 3, 4, 5]
>>> l.index(3)
2
>>> l.index(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 6 is not in list

# 指定した値に合致する要素数を取得
>>> l.count(4)
1

# リストの要素を逆順に変更
>>> print(l)
[1, 2, 3, 4, 5]
>>> l.reverse()
>>> print(l)
[5, 4, 3, 2, 1]

# リストの要素を昇順に並び替え
>>> l = [4, 5, 2, 1, 3]
>>> print(l)
[4, 5, 2, 1, 3]
>>> l.sort(key=None, reverse=False)
>>> print(l)
[1, 2, 3, 4, 5]

# リストの要素を降順に並び替え
>>> l = [4, 5, 2, 1, 3]
>>> print(l)
[4, 5, 2, 1, 3]
>>> l.sort(key=None, reverse=True)
>>> print(l)
[5, 4, 3, 2, 1]

# 特定の文字で区切ってリストに格納
>>> s = 'Hi, Tom. How are you doing?'
>>> print(s.split(' '))
['Hi,', 'Tom.', 'How', 'are', 'you', 'doing?']
>>> print(s.split('.'))
```

```
['Hi, Tom', ' How are you doing?']
```

3.3 タプル型

- リストとタプルの違いは、リストは各要素に対して値を変更できる（mutable）のに対し、タプルは値を変更できない（immutable）。
- その為、タプルは値を一度入れたら書き換えられたくない時など、読み込み専用の用途で利用する。

タプル型

```
# タプルの宣言（括弧ありでもなしでもタプルとして定義される）
>>> t = (1, 2, 3, 4, 5)
>>> print(t)
(1, 2, 3, 4, 5)

>>> t2 = 1, 2, 3
>>> print(t2)
(1, 2, 3)

# タプルの宣言(定義の最後に「,」があるとタプルとして定義される為注意)
# 要素数が一つのみの場合にも最後に「,」を記述する必要がある。
>>> t3 = 1,
>>> t3
(1,)
>>> type(t3)
<class  'tuple'>

# タプルはリストと違い、値を変更できない
>>> t[0] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

# タプルにリストを格納し、そのリストの値は変更可能
>>> t = ([1, 2, 3], [4, 5, 6])
>>> t
([1, 2, 3], [4, 5, 6])
>>> t[0][0]
1
>>> t[0][0] = 100
>>> t
([100, 2, 3], [4, 5, 6])
```

タプルのアンパッキング

```
# タプルの値を変数に代入
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> x, y, z = t
>>> print(x, y, z)
1 2 3

# 特定の変数の値を入れ替える場合等に有効
>>> X = 100
>>> Y = 200
>>> print(X, Y)
100 200
>>> X, Y = Y, X
>>> print(X, Y)
200 100
```

3.4 辞書型（ディクショナリ型）

辞書型（ディクショナリ型）

```
# 辞書型(ディクショナリ型)の宣言 パターン 1
>>> d = {'x': 10, 'y': 20, 'z': 30}
>>> d
{'x': 10, 'y': 20, 'z': 30}

# 辞書型(ディクショナリ型)の宣言 パターン 2
>>> dict(a=100, b=200, c=300)
{'a': 100, 'b': 200, 'c': 300}

# 辞書型(ディクショナリ型)の特定のキーの値の取得
>>> d['y']
20

# 辞書型(ディクショナリ型)にキーと値の追加
>>> d['a'] = 40
>>> d
{'x': 10, 'y': 20, 'z': 30, 'a': 40}

# 辞書型(ディクショナリ型)の特定のキーの削除
>>> del d['a']
>>> d
{'x': 10, 'y': 20, 'z': 30}
```

3.5 辞書型（ディクショナリ型）の操作

辞書型（ディクショナリ型）の操作

```
# 辞書型(ディクショナリ型)のキーリストの取得
>>> d
{'x': 10, 'y': 20, 'z': 30}
>>> d.keys()
dict_keys(['x', 'y', 'z'])
>>> list(d.keys())
['x', 'y', 'z']

# 辞書型(ディクショナリ型)の更新(結合)
# キーが存在している項目は値の更新､キーが存在していない項目はキーと値の追加
>>> d
{'x': 10, 'y': 20, 'z': 30}
>>> d2
{'x': 100, 'a': 1, 'b': 2}
>>> d.update(d2)
>>> d
{'x': 100, 'y': 20, 'z': 30, 'a': 1, 'b': 2}

# 特定のキーの値を取得
>>> d.get('x')
100

# 特定のキーの値を取得(取り出したキーと値は辞書定義からはなくなる)
>>> d
{'x': 100, 'y': 20, 'z': 30, 'a': 1, 'b': 2}
>>> d.pop('a')
1
>>> d.pop('b')
2
>>> d
{'x': 100, 'y': 20, 'z': 30}

# 特定のキーが辞書に含まれるか確認
>>> d
{'x': 100, 'y': 20, 'z': 30}
>>> 'x' in d
True
```

3.6 集合

- リストやタプルとの違いは、集合は要素の重複を許さず、要素の順番を保持しない。
- 組み込み型の集合型には、set 型と frozenset 型の二種類がある。

set 型

```
>>> items = {'note', 'notebook', 'pen'}
>>> type(items)
<class  'set'>
>>> items
{'notebook', 'note', 'pen'}

# 重複している要素は 1つになる
>>> items = {'note', 'notebook', 'pen', 'pen', 'note'}
>>> items
{'notebook', 'note', 'pen'}

# 要素の追加
>>> items.add('book')
>>> items
{'notebook', 'note', 'pen', 'book'}

# 要素の削除
>>> items.remove('pen')
>>> items
{'notebook', 'note', 'book'}

# 要素を取り出して集合から削除
# 順序がない為､取り出される要素は不定
>>> items.pop()
'notebook'
>>> items
{'note', 'book'}
```

- frozenset 型は、set 型を不変にした型 (不変な集合を扱う型)

frozenset 型

```
>>> items = frozenset(['note', 'notebook', 'pen'])
>>> type(items)
<class  'frozenset'>

>>> items
frozenset({'notebook', 'note', 'pen'})
```

```
# 不変な型の為変更はできない
>>> items.add('book')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

集合の演算

```
>>> set_a = {'note', 'notebook', 'pen'}
>>> set_b = {'note', 'book', 'file'}

# 和集合
>>> set_a | set_b
{'notebook', 'note', 'file', 'book', 'pen'}
>>> set_a.union(set_b)
{'notebook', 'note', 'file', 'book', 'pen'}

# 差集合
>>> set_a - set_b
{'notebook', 'pen'}
>>> set_a.difference(set_b)
{'notebook', 'pen'}

# 積集合
>>> set_a & set_b
{'note'}
>>> set_a.intersection(set_b)
{'note'}

# 対称差
>>> set_a ^ set_b
{'notebook', 'book', 'file', 'pen'}
>>> set_a.symmetric_difference(set_b)
{'notebook', 'book', 'file', 'pen'}

# 部分集合か判定
>>> {'note', 'pen'} <= set_a
True
>>> {'note', 'pen'}.issubset(set_a)
True
```

3.7 内包表記

- 内包表記は、リストや集合、辞書等を生成できる構文。

リスト内包表記 (リストの作成)

```
# for 文を使ったリストの作成
>>> numbers = []
>>> for i in range(10):
...     numbers.append(str(i))
...
>>> numbers
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

# リスト内包表記を使ったリストの作成
>>> [str(v) for v in range(10)]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

# ネストしたリストの内包表記
# for 文を使った記述
>>> tuples = []
>>> for x in [1, 2, 3]:
...     for y in [4, 5, 6]:
...         tuples.append((x, y))
...
>>> tuples
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]

# リスト内包表記を使ったネストしたリストの作成
>>> [(x, y) for x in [1, 2, 3] for y in [4, 5, 6]]
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]

# if 文のある内包表記
# for 文を使った記述
>>> even = []
>>> for i in range(10):
...     if i % 2 == 0:
...         even.append(i)
...
>>> even
[0, 2, 4, 6, 8]

# 内包表記を使った記述
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
```

4 関　数

4.1 関数の定義

関数の定義

```
# 関数の定義(引数なし)
>>> def say_hello():
...     print('Hello')
...
>>> say_hello()
Hello

# 関数の定義(引数あり)
>>> def say_something(str):
...     print(str)
...
>>> say_something('Good morning')
Good morning

# 関数の定義(引数あり、デフォルト引数指定あり)
>>> def say_something(str='Hi!'):
...     print(str)
...
# 引数を指定しない場合は、デフォルト引数が使われる
>>> say_something()
Hi!
# 引数を指定した場合は、その引数が使われる。
>>> say_something('Hello')
Hello

# 関数の定義(戻り値あり)
>>> def increment(num):
...     return num + 1
...
>>> increment(1)
2
```

- 関数の中で、return 文が実行されるとそこで処理が終了される為、その後の処理は実行されない。
- return がない関数の戻り値は、None となる。

4.2 関数の引数

関数の引数

```
# 位置引数 -> 関数呼び出し時に渡された順序通りに変数に格納される（引数の数が異なるとエラー）
>>> def increment(num1, num2):
...     return num1 + num2
...
>>> increment(2, 4)
6

# キーワード引数 -> 呼び出し時にキーワード引数を指定（順序は関係ない）
>>> def greeting(name, str):
...     print(str + ', ' + name + '!')
...
>>> greeting(str='Hello', name='Tom')
Hello, Tom!

# デフォルト引数 -> 仮引数にデフォルト値を指定可能(ただし、デフォルト値のある仮引数は、デフォルト値
    のない仮引数よりも後に記述する必要がある。)
>>> def greeting(name, str='Hi'):
...     print(str + ', ' + name + '!')
...
>>> greeting('Tom')
Hi, Tom!

# 可変長の位置引数 -> 任意の数の引数を受け取ることが可能(指定位置は、位置引数の最後で、デフォルト値
    のある引数よりも前)
>>> def say_something(name, *args):
...     for str in args:
...         print("I'm " + name + ". " + str + "!")
...
>>> say_something('Tom', 'Hello', 'Hi', 'Good morning')
I'm Tom. Hello!
I'm Tom. Hi!
I'm Tom. Good morning!

# 可変長のキーワード引数 -> 仮引数に割当てられなかったキーワード引数を辞書型で受け取る(指定位置は
    、一番最後)
>>> def print_birthday(family_name, **kwargs):
...     print("Birthday of " + family_name + " family")
...     for key, value in kwargs.items():
...         print(f'{key}: {value}')
...
>>> print_birthday('Smith', Nancy='1990/1/1', Tom='1993/1/1', Julia='2010/1/1')
```

```
Birthday of Smith family
Nancy: 1990/1/1
Tom: 1993/1/1
Julia: 2010/1/1

# 可変長の位置引数とキーワード引数 -> どのような引数の呼び出しにも対応可能
>>> def say_something(*args, **kwargs):
...     for s in args:
...         print(s)
...     for key, value in kwargs.items():
...         print(f'{key}: {value}')
...
>>> say_something('Hello', 'Hi', 'Bye', Nancy='29 years old', Tom='26 years old')
Hello
Hi
Bye
Nancy: 29 years old
Tom: 26 years old

# キーワードのみ引数 -> 呼び出し時に仮引数名が必須になる引数
#「*」以降がキーワードのみ引数となる
>>> def increment(num, lsat, *, ignore_error=False):
...     pass
...
# 位置引数ではエラーとなる
>>> increment(1, 2, True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: increment() takes 2 positional arguments but 3 were given
# キーワード引数でのみ指定できる
>>> increment(1, 2, ignore_error=True)
>>>

# 位置のみ引数 -> 呼び出し時に仮引数名を指定できない引数
#「/」より前が位置のみ引数となる。仮にabs 関数で仮引数名を指定するとエラーになる。
>>> help(abs)
abs(x, /)
    Return the absolute value of the argument.
>>> abs(x=-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes no keyword arguments
```

引数リストのアンパック

```
# リストに格納された値を引数に渡す
>>> def greeting(x, y, z):
...     print(x)
...     print(y)
...     print(z)
...
>>> contents = ['Hello', 'Good morning', 'Bye']
# 関数呼び出し時に「*」演算子でリストから引数を展開する
>>> greeting(*contents)
Hello
Good morning
Bye

# 辞書に格納された値を引数に渡す
>>> def say_something(name, greeting):
...     print(name)
...     print(greeting)
...
>>> dict = {'greeting': 'Hello'}
# 関数呼び出し時に「**」演算子で辞書から引数を展開する
>>> say_something('Julia', **dict)
Julia
Hello
```

4.3 lambda 式

- lambda 式を使うと、1 行で無名関数を作成できる。
- 無名関数は関数の引数として関数オブジェクトを渡す時によく利用される。

lambda 式の構文

```
lambda 引数 1, 引数 2, 引数 3, ...: 戻り値になる式
```

lambda 式の例

```
>>> increment = lambda num1, num2: num1 + num2
>>> increment(1, 2)
3

# 以下と同義
>>> def increment(num1, num2):
...     return num1 + num2
...
>>> increment(1, 2)
3
```

```
# 第一引数の関数が真になるもののみが残る。
>>> nums = ['one', 'two', 'three']
>>> filterd = filter(lambda x: len(x) == 3, nums)
>>> list(filterd)
['one', 'two']
```

4.4 型ヒント

- アノテーションで関数に型情報を付与する。
- コードの保守性を高めるために利用する。また、mypy などの静的解析ツールによる型チェックの利用もできる。
- 実行時に型チェックが行われる訳ではないので、指定した意外の型を使ってもエラーにはならない。

型情報を付与する為の構文

```
def 関数名（arg1: arg1 の型, arg2: arg2 の型, ...） -> 戻り値の型:
    # 関数で実行したい処理
    return 戻り値

>>> def say_something(name: str, age: int) -> str:
...     print(name)
...     print(age)
...     return name
...
>>> say_something('Tom', 29)
Tom
29
'Tom'
```

5 クラス/インスタンス

5.1 クラス定義

クラス定義

```
class クラス名（基底クラス名）:
    def メソッド名（引数 1, 引数 2, ...）:
        メソッドで実行したい処理
        return 戻り値

>>> class Person(object):
...     def say_something(self):
...         print('Hello!!!')
...
```

```
# クラスをインスタンス化
>>> person = Person()
# インスタンスメソッドの呼び出し
>>> person.say_something()
Hello!!!
```

5.2 インスタンスの初期化

インスタンスの初期化

```
 >>> class Person:
...     def __init__(self):
...         print('Init completed!')
...     def say_something(self):
...         print('Hello!!!')
...
# インスタンスを生成した時に、初期化処理（__init__に定義した処理）が呼ばれる
>>> person = Person()
Init completed!
>>> person.say_something()
Hello!!!
```

5.3 プロパティ

- インスタンスメソッドに「@property」を付けると、そのインスタンスメソッドは、「()」を付けずに呼び出せる
- 「@property」が付いたメソッド -> 値の取得時に呼び出される（読み取り専用） -> getter
- 「@メソッド名.setter」 -> 値を代入する時に呼び出される-> setter
- setter のメソッド名には、「@property」を付けたメソッド名をそのまま利用しなくてはならない。

@property(setter)

```
>>> class Person:
...     @property
...     def name(self):
...         return self._name
...     @name.setter
...     def name(self, name):
...         self._name = name
...
>>> person = Person()
>>> person.name = 'Tom'
>>> person.name
'Tom'
```

- アンダースコアから始まる属性（_name）:プライベートな属性であることを表現する。
- アンダースコア 2 つから始まる属性（__name）:名前修飾が行われる。（例：Person クラスの変数「__name」を「_Person__name」に変換する。）サブクラスでの名前衝突を防ぐために利用される。

5.4 クラスの継承

クラスの継承

```
>>> class Person:
...     def greeting(self):
...         print('Hello')
...     def say_something(self):
...         print('I am human')
...
>>> class Man(Person):
...     # メソッドのオーバーライド
...     def say_something(self):
...         print('I am a man')
...
>>> man = Man()
>>> man.say_something()
I am a man
# 基底クラス（Person クラス）のメソッドも利用可能
>>> man.greeting()
Hello
```

6 モジュール/パッケージ/名前空間/スコープ

6.1 モジュール

- コードを記述した .py ファイル
- クラスや関数の定義が記載されたファイルをモジュールと呼ぶ

モジュールの作成（モジュール名：sample.py）

```
1 import sys
2
3 def say_something_upper(s):
4     # 大文字に変換
5     upper_s = s.upper()
6     return upper_s
7
8 str = sys.argv[1]
9 print(say_something_upper(str))
```

モジュール (sample.py) の呼び出し

```
$ python sample.py hello
HELLO
$
```

6.2 直接実行した時のみに動くコードの記述

直接実行した時のみに動くコードの記述

```
1  import sys
2
3  def say_something_upper(s):
4      # 大文字に変換
5      upper_s = s.upper()
6      return upper_s
7
8  def main():
9      str = sys.argv[1]
10     print(say_something_upper(str))
11
12 if __name__ == '__main__':
13     main()
```

- 「if __name__ == '__main__'」の記述がない場合、他のモジュールから import された場合等でも実行されてしまう。
- そのため、スクリプトとして直接実行された場合にのみ処理が呼ばれるように、上記の通り条件文を記載する。
- グローバル変数「__name__」に格納される値は以下の通り
  - 対話モードで実行した場合の変数「__name__」 → 「__main__」
  - スクリプトとして実行した場合の変数「__name__」 → 「__main__」
  - モジュールとして import された場合の変数「__name__」 → 「モジュール名」
- よって、スクリプトとして実行された場合は、条件文が「真」となりモジュールが実行され、モジュールとして import された場合は、条件文は「偽」となり、モジュールは実行されない。

6.3 パッケージの作成

- パッケージを作成するためには、ディレクトリを作成し、そのディレクトリをパッケージとして扱うために、「__init__.py」ファイルを配置する。(空ファイルでも OK)
- 「__init__.py」にはパッケージの初期化コードを記述したり、「all.py」変数をセットしたりすることも可能。

__init__.py

```
# 空ファイルでOK
```

6.4 パッケージ内のモジュールのインポート

python_programming/sample.py

```
1  from python_package import hello
2
3  def say_hello():
4      print(hello.return_hello())
5
6  def main():
7      say_hello()
8
9  if __name__ == '__main__':
10     main()
```

python_programming/python_package/hello.py

```
def return_hello():
    return 'Hello!!!'
```

実行結果

```
$ python sample.py
Hello!!!
$
```

- import の記述方法

import の記述方法

```
# パッケージをimport
import パッケージ名
import sample_package

# パッケージから特定のモジュールをimport(from 指定なし)
import パッケージ名.モジュール名
import sample_package.greeting

# パッケージから特定のモジュールをimport(from 指定あり)
from パッケージ名 import モジュール名
from sample_package import greeting

# パッケージやモジュールから特定の属性だけをimport
```

```
from パッケージ名/モジュール名 import 関数名
from sample_package.greeting import say_something

# import したパッケージ/モジュール/関数に別名を付与
from パッケージ名 import モジュール名 as 別名
from sample_package.greeting import say_something as s
```

6.5 ワイルドカードを利用した複数属性の一括 import と「__all__」

一括 import

```
# ワイルドカード(*)を利用した一括import
from パッケージ名 import *
from sample_package import *
```

- 上記のようにワイルドカードを利用した import は、可読性が低下したり、意図しない名前の上書きによって不具合が発生する可能性があるため、基本的には利用しない。
- パッケージの「__init__.py」のコードが、「__all__」というリストを定義していれば、それは「from パッケージ import *」の際に import すべきモジュールのリストとなる。
- 例として、下記を定義した場合、「from sample_package import *」を実行されても、"morning", "evening", "night"のモジュールのみが import される。

__init__.py

```
__all__ = ["morning", "evening", "night"]
```

前提：sample_package の中に、"morning", "evening", "night"を含む多数のモジュールが存在

7 組み込み関数/特殊メソッド

7.1 組み込み関数

組み込み関数とは、Python に組み込まれている関数で、何も import することなく利用することが可能。

- isinstance() : 第一引数に渡したインスタンスオブジェクトが、第二引数に渡したクラスに属していれば、True を返す。

isinstance()

```
>>> d = {}

# 第一引数はインスタンスオブジェクト
>>> isinstance(d, dict)
True

# 第二引数をタプルにすると複数のクラスで同時に比較
```

```
>>> isinstance(d, (list, int, dict))
True
```

- issubclass() : isinstance() とほぼ同じだが、第一引数にクラスオブジェクトを取る。

issubclass()

```
# 第一引数はクラスオブジェクト
>>> issubclass(dict, object)
True

# bool 型は int 型のサブクラス
>>> issubclass(bool, (list, int, dict))
True
```

- callable() : 呼び出し可能オブジェクトを判定（関数やクラス、メソッド等、() を付けて呼び出せるオブジェクト）

callable()

```
>>> callable(isinstance) # 関数
True
>>> callable(Exception) # クラス
True
>>> callable(''.split) # メソッド
True
```

- getattr() / setattr() / delattr() : オブジェクトの属性を操作

getattr()/setattr()/delattr()

```
# サンプルクラス定義
>>> class Mutable:
...     def __init__(self, attr_map):
...         for k, v in attr_map.items():
...             setattr(self, str(k), v)
...
# m に属性を設定
>>> m = Mutable({'a': 1, 'b': 2})
>>> m.a
1
>>> attr = 'b'

# 値の取得
>>> getattr(m, attr)
2
```

```
# 値の削除
>>> delattr(m, 'a')
>>> m.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Mutable' object has no attribute 'a'

# getattr()はメソッドも取得可能
>>> str = 'python'
>>> instance_method = getattr(str, 'upper')
>>> instance_method()
'PYTHON'
```

- zip() : 複数のイテラブルの要素を同時に返す

zip()

```
# それぞれのイテラブルのi 番目の要素どうしをまとめる
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> zip(a, b)
<zip object at 0x10827a7d0>
>>> list(zip(a, b))
[(1, 4), (2, 5), (3, 6)]

# zip()は一番短いイテラブルの長さまでしか結果を返さない
>>> d = [1, 2, 3]
>>> e = [4, 5, 6, 7, 8]
>>> f = [9, 10]
>>> zip(d, e, f)
<zip object at 0x10827a7d0>
>>> list(zip(d, e, f))
[(1, 4, 9), (2, 5, 10)]

# 一番長いイテラブルに合わせる場合は、標準ライブラリのitertools.zip_longest()関数を利用
>>> from itertools import zip_longest
>>> list(zip_longest(d, e, f, fillvalue=0))
[(1, 4, 9), (2, 5, 10), (3, 6, 0), (0, 7, 0), (0, 8, 0)]
```

- sorted() : イテラブルの要素を並べ替える

sorted()

```
>>> a = [3, 2, 5, 1, 4]
>>> b = [2, 1, 4, 5, 3]
```

```
# list.sort()は自分自身を並べ替える
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]

# sorted()は新しいリストを返す
>>> sorted(b)
[1, 2, 3, 4, 5]
>>> b
[2, 1, 4, 5, 3]

# reverse=True を指定すると逆順になる
>>> sorted(b, reverse=True)
[5, 4, 3, 2, 1]
```

- sorted() では要素同士を直接比較するため、数値と文字列が混在しているとエラーになる
- その他の組み込み関数は以下ご参照
  Python 組み込み関数一覧
  https://docs.python.org/ja/3/library/functions.html

7.2 特殊メソッド

特殊メソッドとは、メソッド名の前後にアンダースコアが二つ（ __ ）ついているメソッドで、Python から暗黙的に呼び出される。

特殊メソッド

```
# クラス「Word」の中に特殊メソッド、「__init__」「__str__」「__len__」を定義
>>> class Word(object):
...     def __init__(self, text):
...         self.text = text
...     def __str__(self):
...         return 'Word!!!'
...     def __len__(self):
...         return len(self.text)

# クラスをインスタンス化
>>> w = Word('test')

# w を文字列として扱おうとすると、暗黙的に「__str__」が呼ばれる
>>> print(w)
Word!!!

# w の長さを取得するために「len」を実行しようとすると、暗黙的に「__len__」が呼ばれる
```

```
>>> print(len(w))
4

# 本来の記述方法(w というインスタンスの text という属性にアクセスし len の値を取得)
>>> print(len(w.text))
4
```

- その他の特殊メソッドは以下ご参照
  Python 特殊メソッド名
  https://docs.python.org/ja/3/reference/datamodel.html#special-method-names

8 ファイル操作と入出力

8.1 open() を使ったファイルの書き込み

open() を使ったファイルの書き込み

```
# 第一引数はファイル名、第二引数はモード
# モード(引数はオプションであり、省略すれば「r」となる)
#「r」:読み込み専用
#「w」:書き出し専用
#「a」:追記(ファイルの末尾への追記)
#「r+」:読み書き両方
>>> f = open('test1.txt', 'w')
>>> f.write('Test!')
>>> f.close()

# print でもファイルへの書き込みを実施することが可能
>>> f = open('test2.txt', 'w')
>>> print('Print!', file=f)
>>> f.close()

# print を使った場合のオプション
>>> f = open('test3.txt', 'w')
>>> print('My', 'name', 'is', 'Tom', sep='###', end='!!!', file=f)
>>> f.close()
>>>
```

内容の表示

```
$ type test1.txt
Test!
$ type test2.txt
Print!
$ type test3.txt
```

```
My###name###is###Tom!!!
$
```

8.2 read() を使ったファイルの読み込み

read() を使ったファイルの読み込み

```
# 事前準備として「test.txt」を準備
>>> f = open('test4.txt', 'w')
>>> print('My', 'name', 'is', 'Tom', end='!\n', file=f)
>>> print('My', 'name', 'is', 'Nancy', end='!\n', file=f)
>>> print('My', 'name', 'is', 'Julia', end='!\n', file=f)
>>> f.close()

# read()を使ってファイルの内容を読み込む
>>> f = open('test4.txt', 'r')
>>> f.read()
'My name is Tom!\nMy name is Nancy!\nMy name is Julia!\n'
>>> f.close()

# readline()を使って一行づつ読み込む
>>> f = open('test4.txt', 'r')
>>> f.readline()
'My name is Tom!\n'
>>> f.readline()
'My name is Nancy!\n'
>>> f.readline()
'My name is Julia!\n'
>>> f.close()
```

8.3 with ステートメントを使ったファイルの操作

with ステートメントを使ったファイルの操作

```
# with open()を使ってファイルを操作するとclose()処理が不要となる。
>>> with open('test4.txt', 'a') as f:
...     print('with open statement!', file=f)
...
>>>
```

内容の表示

```
$ type test4.txt
My name is Tom!
My name is Nancy!
My name is Julia!
```

```
with open statement!
$
```

- open() によってファイル操作を行う場合、close() を実行するまでメモリを使っているため、必ず最後に close() してメモリを開放する必要がある。
- 一方、with open() の場合、with open() 節のインデントの中の処理が終わると、勝手に close() してくれる為、close() 忘れがなくなる。

参考文献/教材

- [O'REILY]『Python チュートリアル』【著者】Guido van Rossum
- [技術評論社]『Python 実践入門』【著者】陶山 嶺
- [Udemy]『現役シリコンバレーエンジニアが教える Python3 入門 + 応用 + アメリカのシリコンバレー流コードスタイル』【作者】酒井潤