

A I 基礎セミナー

第3回 Python の言語仕様

改訂履歴

日付	担当者	内容
2021/05/08	M. Takeda	Git 公開

目次

- (1) はじめに
- (2) 概要
 - (2.1) Python の特徴
 - (2.2) Python の用途
 - (2.3) Python の注目度
 - (2.4) Python のバージョン
 - (2.5) Python のドキュメント
- (3) 文法
 - (3.1) バッカス・ナウア記法 (Backus-Naur form, BNF)
 - (3.2) Python の完全な構文仕様
- (4) 実装例
 - (4.1) 行列の演算
 - (4.2) 面と等高線で3次元のグラフ表示する
 - (4.3) オブジェクト指向での実装例
- (5) 行構造
 - (5.1) 論理行 (Logical lines)
 - (5.2) 物理行 (Physical lines)
 - (5.3) コメント (Comments)
 - (5.4) シェバン (Shebang)
 - (5.5) エンコード宣言 (Encoding declarations)
 - (5.6) 明示的な行継続 (Explicit line joining)
 - (5.7) 非明示的な行継続 (Implicit line joining)
 - (5.8) 空行 (Blank lines)
 - (5.9) 字下げ (Indentation)
 - (5.10) デリミタ (Delimiters)
- (6) 識別子
 - (6.1) 識別子 (Identifiers)
 - (6.2) キーワード (Keywords)
 - (6.3) 識別子についてのルール
 - (6.4) 識別子についてその他
- (7) リテラル
 - (7.1) 文字列リテラル・バイト列リテラル
 - (7.2) 数値リテラル
- (8) 演算子
 - (8.1) 演算子

- (8.2) 式の評価順序
- (8.3) 演算子の優先順位
- (9) 変数とデータ型
 - (9.1) 動的な型付けの言語処理系
 - (9.2) Python に組み込まれているオブジェクト型
 - (9.3) 変数のスコープ
 - (9.4) 配列 (リスト型)
 - (9.5) 配列 (タプル型)
 - (9.6) 辞書型
 - (9.7) 配列 (Numpy.ndarray)
- (10) 単純文
 - (10.1) Python の単純文
- (11) 制御構文
 - (11.1) if 文
 - (11.2) for 文
 - (11.3) while 文
 - (11.4) with 文
 - (11.5) try 文
- (12) 関数
 - (12.1) 関数定義
 - (12.2) 戻り値
 - (12.3) 引数 (仮引数と実引数)
 - (12.4) ラムダ式 (lambda expression)
 - (12.5) 注釈 (annotation)
 - (12.6) 関数の呼び出し
- (13) モジュール
 - (13.1) モジュール (module)
 - (13.2) パッケージ (package)
- (14) オブジェクト指向
 - (14.1) オブジェクト指向とは
 - (14.2) オブジェクト指向での各要素の関連図
 - (14.3) 単一継承と多重継承
 - (14.4) Python プログラムのオブジェクト指向の実装
 - (14.5) Python でのオブジェクト指向の実装例
- (15) 入出力
 - (15.1) 標準出力／標準エラー出力
 - (15.2) 標準入力
 - (15.3) ファイルへのアクセス
 - (15.4) その他のファイルアクセス
- (16) インタープリタ
 - (16.1) インタープリタの起動・終了
 - (16.2) コマンド仕様
 - (16.3) コマンドラインの引数渡し
- (17) 確認問題
- (18) 確認問題回答用紙
- (資料)
 - (資料1) コーディング・スタイル
 - (資料2) リストオブジェクトのメソッド一覧
 - (資料3) グラフ描画での色一覧

(1) はじめに

- ・ 第3回では、記述言語である Python (Version 3.6) の言語仕様をまとめておきます。
第4回以降は、実装を見ながらセミナーをすすめてゆきますので、
本資料にざっと目をとっておくと良いでしょう。
- ・ 確認問題は、Python の言語仕様の概要が把握できるように設定してあります。
確認問題に目を通して、不明点を本文で確認するような読み方でもよいでしょう。

(2) 概要

- ・ Pythonは、初心者にお勧め出来る言語の一つであり、米国の大学で言語学習の導入段階において最も学ばれています。デスクトップアプリやWebアプリケーション等を構築する際に使用され、データマイニングに最適なツールを備えています。さらに、Pythonはコンピュータサイエンスやデータ解析、生物情報学等、アカデミックなコミュニティでも人気があります。

(2.1) Python の特徴

- ・ 「パイソン (Python)」は、強力なスクリプト言語であり、主な特徴は以下のとおりです。
 - (1) 字下げによって文などのかたまりの範囲 (ブロック、for 文や関数定義など) を示すので、可読性が高い (これを「オフサイドルール (Off-side Rule)」と言います)。
 - (2) オブジェクト指向言語であり、その特性を生かせる (再利用性、大規模化など)。

(2.2) Python の用途

- ・ Python は、特に人口知能 (AI)、マシンラーニング (機械学習)、ディープラーニング (深層学習) の分野で注目の高い言語ですが、それだけでなく、以下の様々な開発に用いられています。
 - (1) WEBアプリケーションの開発
 - (2) GUIアプリケーションの開発
 - (3) デスクトップアプリケーションの開発
 - (4) 人工知能の開発
 - (5) 統計解析アプリケーション制作
 - (6) 組み込みソフトウェアの開発
 - (7) Webスクレイピング・クローリング
- ・ Python を使用した開発例には、以下の様な有名なものがあります：
 - (1) Dropbox (ドロップボックス)・・・Web アプリケーション
 - (2) Dropbox のクライアントアプリ・・・GUIアプリケーション
 - (3) Instagram (インスタグラム)・・・Web アプリケーション
 - (4) ペッパー (Pepper, ソフトバンク)・・・人工知能

「ペッパー (Pepper)」くん



【出典・参考】

Python⇒ <https://techacademy.jp/magazine/8735#sec1>

Python⇒ <https://hajipro.com/python/python-ability>

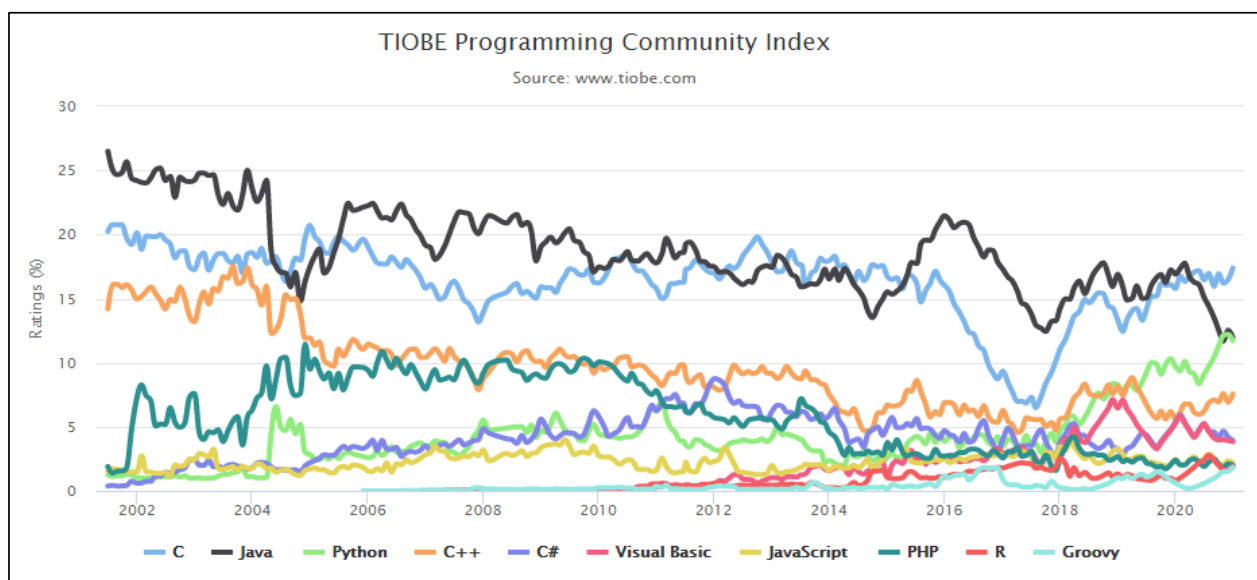
ペッパーくん⇒ <https://www.softbank.jp/robot/consumer/products/>

(2.3) Python の注目度

- ・「TIOBEプログラミングコミュニティインデックス」は、プログラミング言語の人気指標で、インデックスは月に1回更新されます。2020年1月24日時点での上位10位までは以下のようになっています。人気度では Java と並んでいますね・・・

Jan 2021	Jan 2020	Change	Programming Language	Ratings	Change
1	2	▲	C	17.38%	+1.61%
2	1	▼	Java	11.96%	-4.93%
3	3		Python	11.72%	+2.01%
4	4		C++	7.56%	+1.99%
5	5		C#	3.95%	-1.40%
6	6		Visual Basic	3.84%	-1.44%
7	7		JavaScript	2.20%	-0.25%
8	8		PHP	1.99%	-0.41%
9	18	▲▲	R	1.90%	+1.10%
10	23	▲▲	Groovy	1.84%	+1.23%

- ・「TIOBEプログラミングコミュニティインデックス」の時系列は以下のようになっています。ここ数年で人気度が急上昇していることが分かります。



【出典・参考】

⇒ <https://www.tiobe.com/tiobe-index/>

⇒ <https://www.tiobe.com/tiobe-index/programming-languages-definition/>

(2.4) Python のバージョン

- ・ Python は、「元々はAmoebaの使用言語であるABC言語に例外処理やオブジェクト指向を対応させるために作られた言語」で、1991年にヴァンロッサムがPython 0.90のソースコードを公開しました。
- ・ その後、2008年、Python 3.0が公開されましたが、バージョン3.5 迄は 2020年末で既にサポート期限が切れています。
- ・ 3.6 以降のリリース日とサポート期限は以下のようになっています：

バージョン	リリース日	サポート期限
3.6	2016年12月23日	2021年12月
3.7	2018年6月27日	2023年6月
3.8	2019年10月14日	2024年10月
3.9	2020年10月5日	2025年10月

⇒ <https://ja.wikipedia.org/wiki/Python>

(2.5) Python のドキュメント

- ・ Python のドキュメントは以下のURLで、言語とバージョン毎にドキュメントが提供されています：
⇒ <https://docs.python.org/>
- ・ 以下のサイトは入門者にとって分かりやすい Python 言語仕様のドキュメントになっています：
⇒ <https://docs.python.org/ja/3.6/tutorial/index.html>
⇒ http://www.k-techlabo.org/www_python/python_main.pdf
- ・ Python は「初心者にお勧め出来る言語の一つ」といいながら、実際にはかなり高度なことが出来ます。サイト「Python 3.6.6 ドキュメント (<https://docs.python.org/ja/3.6/>)」にはドキュメント一覧のリンク集がありますが、ここに表化して再掲しておきます。

No.	タイトル	URL
1	Python 3.6.6 ドキュメント	https://docs.python.org/ja/3.6/
2	チュートリアル	https://docs.python.org/ja/3.6/tutorial/index.html
3	ライブラリーリファレンス	https://docs.python.org/ja/3.6/library/index.html
4	言語リファレンス	https://docs.python.org/ja/3.6/reference/index.html
5	Python のセットアップと利用	https://docs.python.org/ja/3.6/using/index.html
6	Python HOWTO	https://docs.python.org/ja/3.6/howto/index.html
7	Python モジュールのインストール	https://docs.python.org/ja/3.6/installing/index.html
8	Python モジュールの配布	https://docs.python.org/ja/3.6/distributing/index.html
9	拡張と埋め込み	https://docs.python.org/ja/3.6/extending/index.html
10	Python/C API	https://docs.python.org/ja/3.6/c-api/index.html
11	FAQ	https://docs.python.org/ja/3.6/faq/index.html
12	全モジュール索引	https://docs.python.org/ja/3.6/py-modindex.html
13	総索引	https://docs.python.org/ja/3.6/genindex.html
14	用語集	https://docs.python.org/ja/3/glossary.html

(3) 文法

これ以降では、バージョン3.6 の Python 言語仕様の要点をまとめておきます。

Python の構文とその記法について、下記サイト

⇒ <https://docs.python.org/ja/3.6/reference/introduction.html#notation>

⇒ <https://docs.python.org/ja/3.6/reference/grammar.html>

などを参照・引用しながら、その要約を記します。

(3.1) バッカス・ナウア記法 (Backus-Naur form, BNF)

・これ以降では、「バッカス・ナウア記法 (Backus-Naur form, BNF)」に準じた記法で、文法を表記します。

表記	使用目的
label ::=	名前が 'label' の構文ブロックの、構文の記述の開始を示します。 各構文は「label ::=」から始まります。
(垂直線)	垂直線 () は、複数の選択項目を区切って記すときに使います。 この記号は、この記法において最も結合優先度の低い演算子です。 複数の選択項目中で、選択できる項目は1つだけです。
* (アスタリスク)	アスタリスク「*」は、直前にくる要素のゼロ個以上の繰り返しを表します。
+ (プラス)	プラス「+」は、直前にくる要素の一個以上の繰り返しを表します。
[] (角括弧)	角括弧「[]」に囲われた字句は、字句がゼロ個か一個出現する（別の言い方をすれば、囲いの中の字句はオプションである）ことを示します。
() (丸括弧)	字句のグループ化には丸括弧「()」を使います。
'リテラル文字列'	リテラル文字列はシングルクォート「'」で囲われます。
… (3つのドット)	三つのドットで区切られている二つのリテラル文字は、二つの文字の ASCII 文字コードにおける（包含的な）範囲から文字を一字選ぶことを示します。
各カッコ中の字句<...>	各カッコ中の字句「<...>」は、定義済みのシンボルを記述する非形式的なやりかたです。

(3.2) Python の完全な構文仕様

- ・ 以下は、Python の完全な構文を BNF で記したものです。

(構文) # Grammar for Python

```
# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
#     single_input is a single interactive statement;
#     file_input is a module or sequence of commands read from an input file;
#     eval_input is the input for the eval() functions.
# NB: compound_stmt in single_input is followed by extra NEWLINE!

single_input ::= NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input ::= (NEWLINE | stmt)* ENDMARKER
eval_input ::= testlist NEWLINE* ENDMARKER

decorator ::= '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators ::= decorator+
decorated ::= decorators (classdef | funcdef | async_funcdef)

async_funcdef ::= ASYNC funcdef
funcdef ::= 'def' NAME parameters ['>' test] ':' suite

parameters ::= '(' [typedarglist] ')

typedarglist ::= (tfpdef ['=' test] (',' tfpdef ['=' test])* [',' [
    '*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [',' ]]]
    | '**' tfpdef [',' ]])
    | '*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [',' ]]]
    | '**' tfpdef [',' ]
tfpdef ::= NAME [':' test]

vararglist ::= (vfpdef ['=' test] (',' vfpdef ['=' test])* [',' [
    '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',' ]]]
    | '**' vfpdef [',' ]])
    | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',' ]]]
    | '**' vfpdef [',' ]
)
vfpdef ::= NAME

stmt ::= simple_stmt | compound_stmt
simple_stmt ::= small_stmt ( ';' small_stmt )* [ ';' ] NEWLINE

small_stmt ::= (expr_stmt | del_stmt | pass_stmt | flow_stmt |
    import_stmt | global_stmt | nonlocal_stmt | assert_stmt)

expr_stmt ::= testlist_star_expr (annassign | augassign (yield_expr|testlist) |
    ('=' (yield_expr|testlist_star_expr))* )

annassign ::= ':' test ['=' test]
testlist_star_expr ::= (test|star_expr) (',' (test|star_expr))* [',' ]
```



```

augassign::= ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |
              '<<=' | '>>=' | '**=' | '//=')

del_stmt::= 'del' exprlist
pass_stmt::= 'pass'
flow_stmt::= break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt::= 'break'
continue_stmt::= 'continue'
return_stmt::= 'return' [testlist]
yield_stmt::= yield_expr
raise_stmt::= 'raise' [test ['from' test]]
import_stmt::= import_name | import_from
import_name::= 'import' dotted_as_names

import_from::= ('from' (('.' | '...')* dotted_name | ('.' | '...')+
                    'import' ('*' | '(' import_as_names ')' | import_as_names))

import_as_name::= NAME ['as' NAME]
dotted_as_name::= dotted_name ['as' NAME]
import_as_names::= import_as_name (',' import_as_name)* [' ','']
dotted_as_names::= dotted_as_name (',' dotted_as_name)*
dotted_name::= NAME ( '.' NAME)*
global_stmt::= 'global' NAME (',' NAME)*
nonlocal_stmt::= 'nonlocal' NAME (',' NAME)*
assert_stmt::= 'assert' test [',' test]

compound_stmt::= if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef
                | classdef | decorated | async_stmt

async_stmt::= ASYNC (funcdef | with_stmt | for_stmt)
if_stmt::= 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt::= 'while' test ':' suite ['else' ':' suite]
for_stmt::= 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]

try_stmt::= ('try' ':' suite
            ((except_clause ':' suite)+
             ['else' ':' suite]
             ['finally' ':' suite] |
             'finally' ':' suite))

with_stmt::= 'with' with_item (',' with_item)* ':' suite
with_item::= test ['as' expr]

except_clause::= 'except' [test ['as' NAME]]
suite::= simple_stmt | NEWLINE INDENT stmt+ DEDENT

test::= or_test ['if' or_test 'else' test] | lambdadef
test_nocond::= or_test | lambdadef_nocond
lambdadef::= 'lambda' [vararglist] ':' test
lambdadef_nocond::= 'lambda' [vararglist] ':' test_nocond
or_test::= and_test ('or' and_test)*

```

```

and_test ::= not_test ('and' not_test)*
not_test ::= 'not' not_test | comparison
comparison ::= expr (comp_op expr)*

comp_op ::= '<' | '>' | '==' | '>=' | '<=' | '<>' | '!=' | 'in' | 'not in' | 'is' | 'is not'
star_expr ::= '*' expr
expr ::= xor_expr ('|' xor_expr)*
xor_expr ::= and_expr ('^' and_expr)*
and_expr ::= shift_expr ('&' shift_expr)*
shift_expr ::= arith_expr (('<<' | '>>') arith_expr)*
arith_expr ::= term (('+' | '-') term)*
term ::= factor (('*' | '@' | '/' | '%' | '//') factor)*
factor ::= ('+' | '-' | '~') factor | power
power ::= atom_expr ['**' factor]
atom_expr ::= [AWAIT] atom trailer*

atom ::= ('(' [yield_expr|testlist_comp] ')') |
        '[' [testlist_comp] ']' |
        '{' [dictorsetmaker] '}' |
        NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')

testlist_comp ::= (test|star_expr) (comp_for | (',' (test|star_expr))* [','])
trailer ::= '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist ::= subscript (',' subscript)* [',']
subscript ::= test | [test] ':' [test] [sliceop]
sliceop ::= ':' [test]
exprlist ::= (expr|star_expr) (',' (expr|star_expr))* [',']
testlist ::= test (',' test)* [',']

dictorsetmaker ::= ( ((test ':' test | '**' expr)
                     (comp_for | (',' (test ':' test | '**' expr))* [','])) |
                   ((test | star_expr)
                     (comp_for | (',' (test | star_expr))* [','])) )

classdef ::= 'class' NAME ['(' [arglist] ')'] ':' suite

arglist ::= argument (',' argument)* [',']

argument ::= ( test [comp_for] |
              test '=' test |
              '**' test |
              '*' test )

comp_iter ::= comp_for | comp_if
comp_for ::= [ASYNC] 'for' exprlist 'in' or_test [comp_iter]
comp_if ::= 'if' test_nocond [comp_iter]
encoding_decl ::= NAME

yield_expr ::= 'yield' [yield_arg]
yield_arg ::= 'from' test | testlist

```

(4) 実装例

初めに、Python での実装とその実行結果の例を幾つか示します。実装の感覚をつかんでおきましょう・・・

(4.1) 行列の演算

```
(実装例) *****
# リスト03-(04)-1_行列の演算
*****
import numpy as np

# 行列 X(2行3列)、Y(2行3列) の定義
print("・行列 X(2行3列)、Y(2行3列) の定義")
x = np.array([[11, 12, 13], [21, 22, 23]])
y = np.array([[101, 102, 103], [201, 202, 203]])

gyou, retsu = x.shape
print(x, "\n", type(x), "x : 行=" + str(gyou) + " 列=" + str(retsu))
gyou, retsu = y.shape
print(y, "\n", type(y), "y : 行=" + str(gyou) + " 列=" + str(retsu))

# 行列の和
print("\n・行列の和")
gyou, retsu = (x+y).shape
print(x+y, "\n", type(x+y), "x+y : 行=" + str(gyou) + " 列=" + str(retsu))

# 行列の差
print("\n・行列の差")
gyou, retsu = (x-y).shape
print(x-y, "\n", type(x-y), "x-y : 行=" + str(gyou) + " 列=" + str(retsu))

# 行列の要素同士の積：アダマール積 (Hadamard product)
print("\n・行列の要素同士の積：アダマール積 (Hadamard product)")
gyou, retsu = (x*y).shape
print(x*y, "\n", type(x*y), "x*y : 行=" + str(gyou) + " 列=" + str(retsu))

# 行列の要素同士の割り算
print("\n・行列の要素同士の割り算")
gyou, retsu = (x/y).shape
print(x/y, "\n", type(x/y), "x/y : 行=" + str(gyou) + " 列=" + str(retsu))
```

```
(実行結果) ・行列 X(2行3列)、Y(2行3列) の定義
[[11 12 13]
 [21 22 23]]
<class 'numpy.ndarray'> x : 行=2 列=3
[[101 102 103]
 [201 202 203]]
<class 'numpy.ndarray'> y : 行=2 列=3

・行列の和
[[112 114 116]
 [222 224 226]]
<class 'numpy.ndarray'> x+y : 行=2 列=3

・行列の差
[[ -90 -90 -90]
 [-180 -180 -180]]
<class 'numpy.ndarray'> x-y : 行=2 列=3

・行列の要素同士の積：アダマール積 (Hadamard product)
[[1111 1224 1339]
 [4221 4444 4669]]
<class 'numpy.ndarray'> x*y : 行=2 列=3

・行列の要素同士の割り算
[[0.10891089 0.11764706 0.12621359]
 [0.10447761 0.10891089 0.11330049]]
<class 'numpy.ndarray'> x/y : 行=2 列=3
```

(4.2) 面と等高線で3次元のグラフ表示する

- ・ 以下は、 (x, y) の関数 $Z(x, y)$ を、面と等高線で3次元のグラフ表示する例です。

$$Z(x, y) = \exp(-(x^2 + y^2))$$

```
(実装例) *****
# リスト03-(04)-2_面と等高線で3次元のグラフ表示を行う
# *****
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline

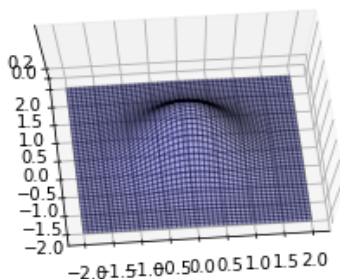
# 関数
def f3(x, y):
    ans = np.exp(-x**2 - y**2)
    return ans

# データ作成
cnt = 50
xlist = np.linspace(-2, 2, cnt)
ylist = np.linspace(-2, 2, cnt)
ztbl = np.zeros((len(xlist), len(ylist)))
for ix in range(cnt):
    for iy in range(cnt):
        ztbl[iy, ix] = f3(xlist[ix], ylist[iy])

# 描画：面表示
xMeshgrid, yMeshgrid = np.meshgrid(xlist, ylist)
plt.figure(figsize=(5, 3.5))
ax = plt.subplot(1, 1, 1, projection='3d')
ax.plot_surface(xMeshgrid, yMeshgrid, ztbl, rstride=1, cstride=1,
                alpha=0.3, color='blue', edgecolor='black')
ax.set_zticks((0, 0.2))
ax.view_init(75, -95)
print(ax, "type(ax)= ", type(ax), "type(xMeshgrid)= ", type(xMeshgrid))
plt.show()

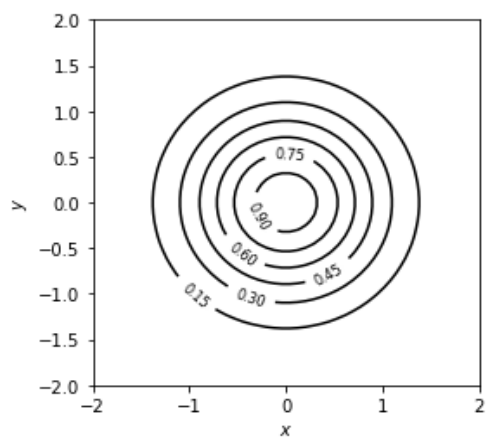
# 描画：等高線表示
xMeshgrid, yMeshgrid = np.meshgrid(xlist, ylist)
plt.figure(1, figsize=(4, 4))
cont = plt.contour(xMeshgrid, yMeshgrid, ztbl, 8, colors='black')
cont.clabel(fmt='%3.2f', fontsize=8)
plt.xlabel('$x$', fontsize=10)
plt.ylabel('$y$', fontsize=10)
print(cont, "type(cont)= ", type(cont))
plt.show()
```

```
(実行結果) Axes3DSubplot(0.125,0.125;0.775x0.755) type(ax)= <class 'matplotlib.axes._subplots.Axes
3DSubplot'> type(xMeshgrid)= <class 'numpy.ndarray'>
```



(実行結果)
続き

```
<matplotlib.contour.QuadContourSet object at 0x00000235C440FD30> type(cont)= <class 'matplotlib.contour.QuadContourSet'>
```



【出典・参考】

⇒ <https://qiita.com/Nezura/items/4a6c56f1ee009ccb0c44>

(4.3) オブジェクト指向での実装例

- ・以下にオブジェクト指向での実装例を示します。

(実装例)

```
#####
# リスト03-(14)-1_オブジェクト指向での実装例
#####
# クラス定義 -----
class AirPlane :
    # クラス変数
    clsName = "飛翔体のクラス"

    # コンストラクタ */
    def __init__( self, name, kind ):
        # インスタンス変数
        self.myName = name
        self.myWingKind = kind

    # インスタンスメソッド
    def Introduce(self, prefix, postfix):
        print(prefix, self.myName, postfix, self.myWingKind, '型の飛翔体', postfix)

    # クラスメソッド
    @classmethod
    def IntroduceClass(self):
        print(' このクラスは', AirPlane.clsName, ' です。')

# 呼出し側 -----

# クラスのインスタンスを作成
plane01 = AirPlane("スーパーマン", "宇宙人");
plane02 = AirPlane("ゼロ戦", "単葉機");
plane03 = AirPlane("スパイダーマン", "超人");

# インスタンスメソッド呼び出し
plane01.Introduce("吾輩は", "である。")
plane02.Introduce("私は", "です。")
plane03.Introduce("我は", "であるぞ。")

# クラス変数のアクセス
print(' クラス名=', AirPlane.clsName )

# クラスメソッド呼び出し
AirPlane.IntroduceClass()
```

(実行結果)

```
吾輩は スーパーマン である。 宇宙人 型の飛翔体 である。
私は ゼロ戦 です。 単葉機 型の飛翔体 です。
我は スパイダーマン であるぞ。 超人 型の飛翔体 であるぞ。
クラス名= 飛翔体のクラス
このクラスは 飛翔体のクラス です。
```

(5) 行構造

Python プログラムは複数の 論理行 (logical lines) から構成されます。

行構造について、下記サイト

⇒ https://docs.python.org/ja/3.6/reference/lexical_analysis.html#line-structure

などを参照・引用しながら、その要約を記します。

(5.1) 論理行 (Logical lines)

- ・ 論理行は一行またはそれ以上の 物理行 (physical line) からなり、物理行の末尾には、明示的または非明示的な 行連結 (line joining) 規則が続きます。
- ・ 論理行の終端は、トークン NEWLINE で表されます。
- ・ 構文上許されている場合 (複合文: compound statement 中の実行文: statement) を除いて、実行文は論理行間にまたがることはできません。

(5.2) 物理行 (Physical lines)

- ・ 物理行とは、行終端コードで区切られた文字列のことです。
- ・ ソースファイルやソース文字列では、各プラットフォーム毎の標準の行終端コードを使用します。
- ・ 行終端コードは、Unix形式ではASCII LF (行送り: linefeed, “\n”) 文字、Windows形式ではASCII 配列の CR LF (復帰: return に続いて行送り, “\r\n”) です。

(5.3) コメント (Comments)

- ・ Pythonのコメントは、文字列リテラル内に入っていないハッシュ文字 (#) で始まり、同じ物理行の末端で終わります。
- ・ 次項のシェバンやエンコード宣言も、文法的にはコメントです。

(例) `# この行は、行末まで全てコメントです!!!`

(5.4) シェバン (Shebang)

- ・ 「シェバン (Shebang)」はスクリプトファイルの第一行目に記述し、「起動してスクリプトを読み込むインタプリタを指定する行」です (bash 等のシェルの記述と同様です)。
- ・ Windowsユーザーがシェバンを書くことは、基本的には必要ありませんが、ファイルごとに実行するPythonのバージョンを変えたい場合や、MacやLinuxのユーザーとコードを共有したい場合は、シェバンを明記します。

【出典・参考】 ⇒ <https://gamasoft.jp/python/python-shebang/>

(例) `#!/usr/bin/env python3`

(5.5) エンコード宣言 (Encoding declarations)

- ・ スクリプトの文字コードは、デフォルトでは「UTF-8」になっています。
 - ・ デフォルト以外のエンコーディングを宣言するには、ファイルの最初の行 (シェバンが第1行の場合は、第2行) として特別なコメント行を追加する必要があります。
- 構文は次のとおりです。

(構文) `# -*- coding: encoding -*-`

(例) `# -*- coding: cp-1252 -*-`

(5.6) 明示的な行継続 (Explicit line joining)

- ・ 二つ以上の物理行を論理行としてつなげるためには、バックスラッシュ文字 (\) を使います。
- ・ バックスラッシュで終わる行にはコメントを入れることはできません。
- ・ バックスラッシュを使ってコメントを継続することはできません。

(例)

```
if 1900 < year < 2100 and 1 <= month <= 12 \
    and 1 <= day <= 31 and 0 <= hour < 24 \
    and 0 <= minute < 60 and 0 <= second < 60:    # Looks like a valid date
    return 1
```

(5.7) 非明示的な行継続 (Implicit line joining)

- ・ 丸括弧 (parentheses)、角括弧 (square bracket)、および波括弧 (curly brace) 内の式は、バックスラッシュを使わずに一行以上の物理行に分割することができます。
- ・ 非明示的に継続された行にはコメントを含めることができます。
- ・ 継続行のインデントは重要ではありません。

(例)

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',      # Dutch names
               'Juli',    'Augustus', 'September', # for the months
               'Oktober', 'November', 'December']  # of the year
```

(5.8) 空行 (Blank lines)

- ・ スペース、タブ、フォームフィード、およびコメントのみを含む論理行は、空行として無視されます。
- ・ 標準的な対話的インタプリタの実装では、完全な空行でできた論理行 (空白文字もコメントも全く含まない空行) は、複数行からなる文の終端を示します。

(5.9) 字下げ (Indentation)

- ・ Python では、字下げによって文などのかたまりの範囲 (ブロック、for 文や関数定義など) を示します。これを「オフサイドルール (Off-side Rule)」と言います。
- ・ 同一ブロック内の各行は、同じ桁数分字下げする必要があります。

(例)

```
def is_even(a):
    if a / 2.0 == a / 2:
        return True
    else:
        return False
```

(5.10) デリミタ (Delimiters)

- ・ 以下のトークンは文法上のデリミタとして働きます。
- ・ 後半の累算代入演算子 (augmented assignment operator) は、字句的にはデリミタとして振舞いますが、演算も行います (詳細は別項参照のこと)。

()	[]	{	}		
.	:	.	;	@	=	->	
+=	-=	*=	/=	//=	%=	@=	
&=	=	^=	>>=	<<=	**=		

- ・ 以下の印字可能 ASCII 文字は、他のトークンの一部として特殊な意味を持っていたり、字句解析器にとって重要な意味を持っています。

'	"	#	¥
---	---	---	---

- ・ 以下の印字可能 ASCII 文字は、Python では、文字列リテラルやコメント以外では使われていません。

\$?	`
----	---	---

(6) 識別子

Python プログラムの識別子とキーワードについて、下記サイト

⇒ https://docs.python.org/ja/3.6/reference/lexical_analysis.html#identifiers

などを参照・引用しながら、その要約を記します。

(6.1) 識別子 (Identifiers)

- ・ Python における識別子 (identifier) の構文は以下のとおりです。
- ・ 識別子の長さには制限がありません。大小文字は区別されます。
- ・ 識別子として使用可能な文字種は、ASCII だけでなく、ASCII 範囲外からも利用可能です。

(構文)

```
identifier ::= xid_start xid_continue*
id_start   ::= <all characters in general categories
               Lu, Ll, Lt, Lm, Lo, Nl, the underscore,
               and characters with the Other_ID_Start property>
id_continue ::= <all characters in id_start, plus characters
               in the categories Mn, Mc, Nd, Pc
               and others with the Other_ID_Continue property>
xid_start  ::= <all characters in id_start
               whose NFKC normalization is in "id_start xid_continue*">
xid_continue ::= <all characters in id_continue
               whose NFKC normalization is in "id_continue*">
```

Lu - 大文字 (uppercase letters)

Ll - 小文字 (lowercase letters)

Lt - 先頭が大文字 (titlecase letters)

Lm - 修飾文字 (modifier letters)

Lo - その他の文字 (other letters)

Nl - 数値を表す文字 (letter numbers)

Mn - 字幅のない記号 (nonspacing marks)

Mc - 字幅のある結合記号 (spacing combining marks)

Nd - 10 進数字 (decimal numbers)

Pc - 連結用句読記号 (connector punctuations)

Other_ID_Start - PropList.txt にある、後方互換性をサポートするための明示的な文字のリスト

Other_ID_Continue - 同様

(6.2) キーワード (Keywords)

- ・以下の識別子は、予約語、または Python 言語における キーワード (keyword) として使われ、通常の識別子として使うことはできません。

and	def	finally	in	or	while
as	del	for	is	pass	with
assert	elif	from	lambda	raise	yield
break	else	global	None	return	
class	except	if	nonlocal	TRUE	
continue	FALSE	import	not	try	

(6.3) 識別子についてのルール

- ・ Python3 では日本語の変数名も使用できます。
然しながら、「Pythonコードのスタイルガイド」では、「英小文字・数とアンダースコア “_” 。最初の文字は数字以外。」という命名規約が推奨されています。
- ・ 変数名をアンダースコア “_” のみとすることも文法的に可能ですが、アンダースコア “_” のみからなる変数名は、Python の習慣として「その変数は不使用」ということを表現しています。

(6.4) 識別子についてその他

- ・ Python の識別子は、オブジェクトに付けられた名前という意味があり、「名前をモジュールが束縛します」というように、オブジェクトが識別子を束縛するという説明がよく出てきます。

(7) リテラル

Python プログラムのリテラルについては、下記サイト

⇒ https://docs.python.org/ja/3.6/reference/lexical_analysis.html#literals

にありますが、その要約などを記します。

(7.1) 文字列リテラル・バイト列リテラル

- ・文字列リテラルとバイト列リテラルは、対応する一重引用符 (') または二重引用符 (") で囲います。
- ・また、対応する三連の一重引用符や二重引用符で囲うこともできます (通常、「三重クオート文字列 (triple-quoted string)」と呼ばれます)。これにより、複数の行に渡る文字列を記述することができます。
- ・バックスラッシュ (\) 文字で、本来特別な意味を持つ文字、例えば改行文字、バックスラッシュ自身、クオート文字などを、エスケープできます。
- ・文字列リテラルとバイト列リテラルの両方は、任意で文字 'r' または 'R' をプレフィックスに持つことができます。そのような文字列は raw strings と呼ばれ、バックスラッシュをリテラル文字として扱います。

(7.1.1) 文字列リテラル (String literals)

```
(構文) stringliteral ::= [stringprefix] (shortstring | longstring)
stringprefix ::= "r" | "u" | "R" | "U" | "f" | "F"
               | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
shortstring ::= """ shortstringitem* """ | ''' shortstringitem* '''
longstring ::= """ longstringitem* """ | """ longstringitem* """
shortstringitem ::= shortstringchar | stringescapeseq
longstringitem ::= longstringchar | stringescapeseq
shortstringchar ::= <any source character except "&quot;" or newline or the quote>
longstringchar ::= <any source character except "&quot;">
stringescapeseq ::= "&quot;" <any source character>
```

(7.1.2) バイト列リテラル (Bytes literals)

- ・バイト列リテラルには、常に 'b' や 'B' が接頭します。これらによって、文字列型ではなく バイト列型のインスタンスが作成されます。
- ・バイト列リテラルは ASCII 文字のみ含むことができます。
- ・128 以上の数値を持つバイトはエスケープして表されなければなりません。

```
(構文) bytesliteral ::= bytesprefix (shortbytes | longbytes)
bytesprefix ::= "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes ::= """ shortbytesitem* """ | ''' shortbytesitem* '''
longbytes ::= """ longbytesitem* """ | """ longbytesitem* """
shortbytesitem ::= shortbyteschar | bytesescapeseq
longbytesitem ::= longbyteschar | bytesescapeseq
shortbyteschar ::= <any ASCII character except "&quot;" or newline or the quote>
longbyteschar ::= <any ASCII character except "&quot;">
bytesescapeseq ::= "&quot;" <any ASCII character>
```

(7.1.3) フォーマット済み文字列リテラル

- ・「フォーマット済み文字列リテラル (formatted string literal)」または f-string は、接頭辞 'f' または 'F' の付いた文字列リテラルです。これらの文字列には、波括弧 {} で区切られた式である置換フィールドを含めることができます。
- ・二重の波括弧 '{{' または '}}' は、それぞれに対応する一重の波括弧に置換されます。
- ・一重の開き波括弧 '{' で、置換フィールド(replacement_field)を開始します。置換フィールドは Python の式で始まり、式の後には感嘆符 '!' に続けて変換フィールド(conversion)を追記し、さらにコロン ':' に続けて書式指定子(format_spec)を追記します。そして置換フィールドは、閉じ波括弧 '}' で終了します。
- ・もし変換フィールドが指定されていた場合、式の評価結果はフォーマットの前に変換されます。変換 '!s' は str() を、 '!r' は repr() を、そして '!a' は ascii() を呼び出します。

(構文)

```
f_string      ::= (literal_char | "{" | "}") | replacement_field)*
replacement_field ::= "{" f_expression ["!" conversion] [":" format_spec] "}"
f_expression   ::= (conditional_expression | "*" or_expr)
                  ("," conditional_expression | "," "*" or_expr)* [","]
                  | yield_expression
conversion     ::= "s" | "r" | "a"
format_spec    ::= (literal_char | NULL | replacement_field)*
literal_char   ::= <any code point except "{", "}" or NULL>
```

(実装例)

```
*****
# リスト03-(07)-1_フォーマット済み文字列リテラル
*****
import decimal
import datetime as dt

name = "Fred"
print( f"He said his name is {name!r}." )

width = 10
precision = 4
value = decimal.Decimal("12.34567")
print( f"result: {value:{width}.{precision}}" )

today = dt.datetime(year=2017, month=1, day=27)
print( f"{today:%B %d, %Y}" )

number = 1024
print( f"{number:#0x}" )
```

(出力結果)

```
He said his name is 'Fred'.
result:      12.35
January 27, 2017
0x400
```

(7.1.4) エスケープシーケンス

- ・ 以下に、文字列リテラル内で認識されるエスケープシーケンスを示します。

エスケープシーケンス	意味	注釈
<code>\newline</code>	バックスラッシュと改行文字が無視されます	
<code>\f</code>	バックスラッシュ (¥)	
<code>\'</code>	一重引用符 (')	
<code>\"</code>	二重引用符 (")	
<code>\a</code>	ASCII 端末ベル (BEL)	
<code>\b</code>	ASCII バックスペース (BS)	
<code>\f</code>	ASCII フォームフィード (FF)	
<code>\n</code>	ASCII 行送り (LF)	
<code>\r</code>	ASCII 復帰 (CR)	
<code>\t</code>	ASCII 水平タブ (TAB)	
<code>\v</code>	ASCII 垂直タブ (VT)	
<code>\ooo</code>	8 進数値 ooo を持つ文字	
<code>\xhh</code>	16 進数値 hh を持つ文字	

- ・ 以下に、文字列でのみ認識されるエスケープシーケンスを示します。

エスケープシーケンス	意味	注釈
<code>\N{name}</code>	Unicode データベース中で name という名前の文字	文字列でのみ認識される
<code>\uxxxx</code>	16-bit の16進値 xxxx を持つ文字	文字列でのみ認識される
<code>\Uxxxxxxxx</code>	32-bit の16進値 xxxxxxxx を持つ文字	文字列でのみ認識される

(7.2) 数値リテラル

- ・数値リテラルには、整数(integer)、浮動小数点数(floating point number)、虚数(imaginary numbers)の3種類あります。
- ・数値リテラルでは、アンダースコアで数字をグループ化(3桁毎、1バイト毎など)して読みやすくなります。アンダースコアは数値の評価では無視されます。アンダースコアは数字と数字の間に1つだけ、又は 0xのような基数指定の直後に1つだけ挿入できます。

(7.2.1) 整数リテラル

- ・値がメモリ上に収まるかどうかという問題を除けば、整数リテラルには長さの制限がありません。
- ・数を表現する際に下表の接頭辞を付けることで、2進法、8進法、16進法の基数指定ができます。

接頭辞	表現	基数	例 (括弧内は10進法での表現)
0b	2 進法	2	0b1011011 (= 91)
0o	8 進法	8	0o1234567 (= 342391)
0x	16 進法	16	0x7b4f (= 31567)

(構文)

```
integer      ::= decinteger | bininteger | octinteger | hexinteger
decinteger   ::= nonzerodigit ([ "_" ] digit)* | "0"+ ([ "_" ] "0")*
bininteger   ::= "0" ("b" | "B") ([ "_" ] bindigit)+
octinteger   ::= "0" ("o" | "O") ([ "_" ] octdigit)+
hexinteger   ::= "0" ("x" | "X") ([ "_" ] hexdigit)+
nonzerodigit ::= "1"... "9"
digit        ::= "0"... "9"
bindigit     ::= "0" | "1"
octdigit     ::= "0"... "7"
hexdigit     ::= digit | "a"... "f" | "A"... "F"
```

(例)

7	2147483647	0o177	0b100110111
3	79228162514264337593543950336	0o377	0xdeadbeef
	100_000_000_000 (3桁区切り)	0b_1110_0101 (4bit区切り)	

(7.2.2) 浮動小数点数リテラル

- ・整数部と指数部は常に 10 を基数として解釈されます。
- 例えば、077e010 は正しい表記であり、77e10 と同じ数を表します。

(構文)

```
floatnumber  ::= pointfloat | exponentfloat
pointfloat   ::= [digitpart] fraction | digitpart "."
exponentfloat ::= (digitpart | pointfloat) exponent
digitpart    ::= digit ([ "_" ] digit)*
fraction     ::= "." digitpart
exponent     ::= ("e" | "E") ["+" | "-"] digitpart
```

(例)

3.14	10.	.001	1e100	3.14e-10	0e0	3.14_15_93 (2桁区切り)
------	-----	------	-------	----------	-----	--------------------

(7.2.3) 虚数 (imaginary) リテラル

- ・虚数リテラルは、実数部が 0.0 の複素数を表します。
- ・複素数は二つ組の浮動小数点型の数値で表され、それぞれの数値は浮動小数点型と同じ定義域の範囲を持ちます。
- ・実数部がゼロでない複素数を生成するには、(3+4j) のように実数部を示す浮動小数点リテラルに、虚数リテラルを '+' で結合します。

(構文)

```
imagnumber ::= (floatnumber | digitpart) ("j" | "J")
```

(例)

3.14j	10.j	10j	.001j	1e100j	3.14e-10j	3.14_15_93j (2桁区切り)
-------	------	-----	-------	--------	-----------	---------------------

(8) 演算子

Python プログラムの演算子について、下記サイト

⇒ <http://www.tohoho-web.com/python/operators.html>

⇒ <https://docs.python.org/ja/3.6/reference/expressions.html>

などを参照・引用しながら、その要約を記します。

(8.1) 演算子

・以下のトークンは演算子です。

+	-	*	**	/	//	%	@
<<	>>	&		^	~		
<	>	<=	>=	==	!=		
+=	-=	*=	/=	//=	%=	@=	
&=	=	^=	>>=	<<=	**=		

(8.1.1) 代数演算子

No.	書式	意味
1	+a	正数
2	-a	負数
3	a + b	加算
4	a - b	減算
5	a * b	乗算
6	a / b	除算
7	a % b	a を b で割った余り
8	a ** b	a の b 乗
9	a // b	a を b で切り捨て除算

(8.1.2) ビット演算子

No.	書式	意味
1	~a	ビット反転
2	a & b	AND: 論理積 (aもbも1のビットが1)
3	a b	OR: 論理和 (aまたはbが1のビットが1)
4	a ^ b	XOR: 排他的論理和 (aまたはbの一方のみ1のビットの場合1、それ以外は0)
5	a << b	a を b ビット分左シフト
6	a >> b	a を b ビット分右シフト

(8.1.3) 代入演算子

No.	書式	意味
1	a = b	a に b を代入する
2	a += b	a = a + b に同じ
3	a -= b	a = a - b に同じ
4	a *= b	a = a * b に同じ
5	a /= b	a = a / b に同じ
6	a %= b	a = a % b に同じ
7	a **= b	a = a ** b に同じ
8	a //= b	a = a // b に同じ
9	a &= b	a = a & b に同じ
10	a = b	a = a b に同じ
11	a ^= b	a = a ^ b に同じ
12	a <<= b	a = a << b に同じ
13	a >>= b	a = a >> b に同じ

(8.1.4) 文字列演算

桁位置は 0 から数えます。n や m にマイナスの数を指定すると、文字列の後ろから数えます。

No.	書式	意味
1	a + b	文字列 a と 文字列 b を連結します
2	n * a	文字列 a を n 回繰り返します
3	a[n]	文字列 a の中の n 番目の文字を取り出します
4	a[n:m]	文字列 a の中の n 番目から m 番目までの部分文字列を取り出します
5	a[n:]	文字列 a の中の n 番目から最後まで部分文字列を取り出します
6	a[:m]	文字列 a の中の 0 番目から m 番目までの部分文字列を取り出します
7	a[n:m:s]	文字列 a の中の n 番目から m 番目までの部分文字列を s個とばして取り出します

(8.1.5) 比較演算子

No.	書式	意味
1	a == b	a が b と等しいならば真。それ以外は偽。
2	a != b	a が b と異なるならば真。それ以外は偽。
3	a < b	a が b よりも小さいならば真。それ以外は偽。
4	a > b	a が b よりも大きいならば真。それ以外は偽。
5	a <= b	a が b 以下ならば真。それ以外は偽。
6	a >= b	a が b 以上ならば真。それ以外は偽。
7	a <> b	a が b と異なるならば真。それ以外は偽。
8	a is b	a が b と等しいならば真。それ以外は偽。
9	a is not b	a が b と異なるならば真。それ以外は偽。
10	a in b	a が b に含まれるならば真。それ以外は偽。
11	a not in b	a が b に含まれないならば真。それ以外は偽。

(8.1.6) ブール演算子

No.	書式	意味
1	a and b	a も b も真であれば真。それ以外は偽。
2	a or b	a または b が真であれば真。それ以外は偽。
3	not a	a が偽であれば真。それ以外は偽。

(8.1.7) 条件演算子

No.	書式	意味
1	x if c else y	c が真であれば x を、さもなければ y を返します。

(8.1.8) ドット演算子


No.	書式	意味
1	x.a	インスタンスx のインスタンス変数 a を参照します。
2	x.m	インスタンスx のインスタンスメソッド m を呼び出します。
3	X.A	クラスX のクラス変数 A を参照します。
4	X.M	クラスX のクラスメソッド M を呼び出します。

(8.2) 式の評価順序

- Python は、式を左から右へと順に評価します。
- ただし、代入式を評価するときは、右辺が左辺よりも先に評価されます。

(8.3) 演算子の優先順位

- ・ Python における演算子の優先順位を下表に示します。
- ・ 優先順位の最も低い（結合が最も弱い）ものから最も高い（結合が最も強い）ものに並べてあります。
- ・ 同じボックス内の演算子の優先順位は同じです。
- ・ 構文が明示的に示されていないものは二項演算子です。

優先度	演算子	説明
(低)  (高)	lambda	ラムダ式
	if -- else	条件式
	or	ブール演算 OR
	and	ブール演算 AND
	not x	ブール演算 NOT
	in, not in, is, is not, <, <=, >, >=, !=, ==	帰属や同一性のテストを含む比較
		ビット単位 OR
	^	ビット単位 XOR
	&	ビット単位 AND
	<<, >>	シフト演算
	+, -	加算および減算
	*, @, /, //, %	乗算, 行列の積, 除算, 切り捨て除算, 剰余
	+x, -x, ~x	正数, 負数, ビット単位 NOT
	**	べき乗 [6]
	await x	Await 式
	x[index], x[index:index], x(arguments...), x.attribute	添字指定, スライス操作, 呼び出し, 属性参照
	(expressions...), [expressions...], {key: value...}, {expressions...}	式結合またはタプル表示, リスト表示, 辞書表示, 集合表示

(9) 変数とデータ型

Python プログラムのデータ型は、オブジェクト型としてまとめられていますが、下記サイト

⇒ <https://docs.python.org/ja/3.6/reference/datamodel.html>

などを参照・引用しながら、その要約を記します。

(9.1) 動的な型付けの言語処理系

- ・ C 言語やJava は「強い型付けの言語処理系」であるのに対し、Python は「動的な型付けの言語処理系」です。
即ち、C 言語やJava は変数の使用に先立って変数の確保とデータ型を宣言する必要があるのに対し、Python では、変数の使用に先立って変数の確保とデータ型を宣言する必要がありません。
- ・ Python では、ある変数に値を設定した後で、別の型の値でその変数の内容を上書きすることも出来ます。
以下にその例を示します。

```
num = 1.2          # 浮動小数点数
num = 1.2e3        # 浮動小数点数(指数表記) 1.2 × 10**3
num = 1.2E-3       # 浮動小数点数(指数表記) 1.2 × 10**-3
num = 123          # 整数 (10進数)
num = 0o173        # 整数 ( 8進数)
num = 0x7B         # 整数 (16進数)
num = 0b01111011   # 整数 ( 2進数)
```

【出典・参考】 ⇒ http://www.k-techlabo.org/www_python/python_main.pdf

(9.2) Python に組み込まれているオブジェクト型

- Python における オブジェクト (object) とは、データを抽象的に表したものです。
- Python プログラムにおけるデータは全て、オブジェクトまたはオブジェクト間の関係として表されます。
- すべての属性は、同一性、型、値をもっています。
 - 「同一性 (identity)」は生成されたあととは変更されません。
 - これはオブジェクトのアドレスのようなものです。
 - 'is' 演算子は2つのオブジェクトの同一性を比較します。id() 関数は同一性を表す整数を返します。
- 以下に Python に組み込まれているオブジェクト型の一覧を示します。

オブジェクト名	説明
None	<ul style="list-style-type: none">このオブジェクトは、値が存在しないことを示します (nullオブジェクトに相当します)。例えば、明示的に値を返さない関数は None を返します。このオブジェクトは、組み込み名 None でアクセスします。
NotImplemented	<ul style="list-style-type: none">被演算子が該当する演算を行うための実装をもたないメソッドは、この値を返します。このオブジェクトは、組み込み名 NotImplemented でアクセスします。
Ellipsis	<ul style="list-style-type: none">「何らかの値が存在するが省略されている」ことを表すオブジェクト名です。例えば、NumPyでの多次元配列のスライス記法で ... (ピリオド三つ)は指定されていない全ての次元について、インデックスが任意であることを示します。このオブジェクトは、リテラル ... または組み込み名 Ellipsis でアクセスします。
numbers.Number	<ul style="list-style-type: none">数値リテラルによって作成されたり、算術演算や組み込みの算術関数によって返される数値を表すオブジェクトです。numbers.Numberは数の集合全体を表すクラスです。Python では整数、浮動小数点数、複素数の間で区別を行っています。<ul style="list-style-type: none">(1) 「numbers.Integral (整数)」は、整数(正の数および負の数)を表す型です。 整数型には「整数 (int)」と「ブール値 (bool、False/True)」の2種類あります。(2) 「numbers.Real (float) (実数)」は、倍精度浮動小数点数を表現します。(3) 「numbers.Complex (複素数)」は、倍精度浮動小数点を 2 つ一組にして複素数を表現します。複素数 z の実数部・虚数部は、<code>z.real</code> および <code>z.imag</code> で参照します。
シーケンス型 (Sequences)	<ul style="list-style-type: none">この型は、有限の順序集合 (ordered set) を表現します。要素は非負の整数でインデクス化されています。組み込み関数 len() を使うと、シーケンスの要素数を返します。シーケンス a の要素 i (i=0~)は <code>a[i]</code>で選択します。シーケンスはスライス操作 (slice) もサポートしています (<code>a[i:j]</code>、<code>a[i:j:k]</code>など)。シーケンス型には、一度生成されるとその値を変更することができない「変更不能なシーケンス (immutable sequence)」と、作成後に変更することができる「変更可能なシーケンス型 (mutable sequence)」の2種類あります。変更不能なシーケンス型には以下の3種類あります。<ul style="list-style-type: none">(1) 文字列型 (string)<ul style="list-style-type: none">文字列はUnicodeコードポイントを表現する値の配列です。Pythonには 文字型が無い代わりに、どの文字も長さ1の文字列型で文字列を表現します。(2) タプル型 (tuple)<ul style="list-style-type: none">タプルの要素は任意の Python オブジェクトです。個々の要素を表現する式をカンマで区切り、全体を丸括弧 () で囲んで構成します。

	<p>(3) bytes</p> <ul style="list-style-type: none"> 要素はバイトで、$0 \leq x < 256$ の範囲の整数で表現される不変な配列です。 バイト列リテラルや組み込みの <code>bytes()</code> コンストラクタを使って本オブジェクトを作成します。 <p>・変更可能なシーケンス型には以下の2種類あります。</p> <p>(1) リスト型 (list)</p> <ul style="list-style-type: none"> リストの要素は任意の Python オブジェクトです。 個々の要素を表現する式をカンマで区切り、全体を角括弧[]で囲んで構成します。 <p>(2) バイト配列 (bytearray)</p> <ul style="list-style-type: none"> 要素はバイトで、$0 \leq x < 256$ の範囲の整数で表現される変更可能な配列です。 組み込みの <code>bytearray()</code> コンストラクタによって作成されます。
集合型 (Set types)	<ul style="list-style-type: none"> 集合型は、順序のない、ユニークで不変なオブジェクトの有限集合を表現します。そのため、(配列の)添字を使ったインデックスアクセスはできません。その代わり、イテレートは可能です。また組み込み関数 <code>len()</code> は集合の要素数を返します。 集合の要素には、辞書のキーと同じルールが適用されます。 組み込み集合型には以下の2種類あります。 <p>(1) 集合型</p> <ul style="list-style-type: none"> 可変な集合型です。組み込みの <code>set()</code> コンストラクタで作成され、後から <code>add()</code> などのいくつかのメソッドで更新できます。 <p>(2) Frozen set 型</p> <ul style="list-style-type: none"> 不変な集合型です。組み込みの <code>frozenset()</code> コンストラクタによって作成されます。 <code>frozenset</code> は不変でハッシュ可能(hashable)なので、別の集合型の要素になったり、辞書のキーにすることができます。
マッピング型 (Mappings)	<ul style="list-style-type: none"> 任意のインデックス集合でインデックス化された、オブジェクトからなる有限の集合を表現します。 添字表記 <code>a[k]</code> は、<code>k</code> でインデックス指定された要素を <code>a</code> から選択します。 組み込み関数 <code>len()</code> は、マッピング内の要素数を返します。 マッピング型には以下の1種類あります。 <p>(1) 辞書型 (Dictionaries)</p> <ul style="list-style-type: none"> ほぼ任意のキー値でインデックスされたオブジェクトからなる有限の集合を表します。 キー (key) として使えない値の型は、リストや辞書、そしてオブジェクトの同一性でなくオブジェクトの値で比較されるその他の変更可能な型です。 辞書は変更可能な型で、<code>{...}</code> 表記で生成します。
呼び出し可能型 (callable type)	<ul style="list-style-type: none"> 関数呼び出し操作(<code>call</code>)を行うことができる型です。 <p>(1) ユーザ定義関数 (user-defined function)</p> <ul style="list-style-type: none"> ユーザ定義関数オブジェクトは、関数定義を行うことで生成されます。関数は、仮引数 (formal parameter) リストと同じ数の要素が入った引数リストとともに呼び出されます。 関数オブジェクトはまた、任意の属性を設定したり取得したりできます。この機能は、例えば関数にメタデータを付与したい場合などに使えます。 <p>(2) インスタンスメソッド</p> <ul style="list-style-type: none"> インスタンスメソッドオブジェクトは、クラス、クラスインスタンスと任意の呼び出し可能オブジェクト (通常はユーザ定義関数) を結びつけます。 メソッドもまた、関数オブジェクトの任意の関数属性にアクセスできます。 ユーザ定義関数オブジェクトからインスタンスメソッドオブジェクトへの変換は、インスタンスから属性が取り出されるたびに行われます。

	<p>(3) ジェネレータ関数 (generator function)</p> <ul style="list-style-type: none"> ・ <code>yield</code> 文を使う関数もしくはメソッドを「ジェネレータ関数」と呼びます。 そのような関数が呼び出されたときは常に、関数の本体を実行するのに使えるイテレータオブジェクトを返します。 <p>(4) コルーチン関数 (coroutine function)</p> <ul style="list-style-type: none"> ・ <code>async def</code> を使用して定義された関数やメソッドを コルーチン関数と呼びます。 そのような関数が呼び出された時、coroutine オブジェクトを返します。 <p>(5) 非同期ジェネレータ関数 (asynchronous generator function)</p> <ul style="list-style-type: none"> ・ <code>async def</code> を使って定義され、<code>yield</code> 文を使用している関数やメソッドを非同期ジェネレータ関数と呼びます。 そのような関数は、呼び出されたとき、非同期イテレータオブジェクトを返します。 <p>(6) 組み込み関数 (built-in function)</p> <ul style="list-style-type: none"> ・ 組み込み関数オブジェクトはC関数へのラッパーです。(例: <code>len()</code> や <code>math.sin()</code>) 組み込み関数の引数の数や型は C 関数で決定されています。 <p>(7) 組み込みメソッド (built-in method)</p> <ul style="list-style-type: none"> ・ 組み込み関数を別の形で隠蔽したもので、こちらの場合には C 関数に渡される何らかのオブジェクトを非明示的な外部引数として持っています。 <p>(8) クラス (Class)</p> <ul style="list-style-type: none"> ・ クラスのオブジェクトは通常、そのクラスの新たなインスタンスのファクトリとして振舞いますが、<code>__new__()</code> をオーバーライドして、バリエーションを持たせられます。 <p>(9) クラスのインスタンス</p> <ul style="list-style-type: none"> ・ 任意のクラスのインスタンスは、クラスで <code>__call__()</code> メソッドを定義することで呼び出し可能になります。
モジュール (module)	<ul style="list-style-type: none"> ・ モジュールは Python コードの基礎的な組織単位で、<code>import</code> 文あるいは <code>importlib.import_module()</code> や組み込みの <code>__import__()</code> のような関数を呼び出すことで起動される import system によって作成されます。
カスタムクラス型	<ul style="list-style-type: none"> ・ カスタムクラス型は通常、クラス定義で生成されます。 ・ クラス定義は、クラスオブジェクトを定義します（構文を参照）。 <p>構文</p> <pre>classdef ::= [decorators] "class" classname [inheritance] ":" suite inheritance ::= "(" [argument_list] ")" classname ::= identifier</pre>
クラスインスタンス (class instance)	<ul style="list-style-type: none"> ・ クラスインスタンスは、クラスオブジェクトを呼び出して生成します。
I/O オブジェクト (ファイルオブジェクトの別名)	<ul style="list-style-type: none"> ・ <code>file object</code> は開かれたファイルを表します。 ・ ファイルオブジェクトを作るための様々なショートカットがあります。 (<code>open</code>、<code>os.popen</code>、<code>os.fdopen</code>、ソケットオブジェクトの <code>makefile()</code> メソッド等) ・ オブジェクト <code>sys.stdin</code>、<code>sys.stdout</code> および <code>sys.stderr</code> は、インタプリタの標準入力、標準出力、および標準エラー出力ストリームに対応するファイルオブジェクトです。
内部型 (internal type)	<ul style="list-style-type: none"> ・ インタプリタが内部的に使っている以下の型は、ユーザに公開されています。 <p>(1) コードオブジェクト</p> <ul style="list-style-type: none"> ・ コードオブジェクトは バイトコンパイルされた (byte-compiled) 実行可能な Python

コード、別名バイトコード(bytecode) を表現します。

(2) フレーム (frame) オブジェクト

- ・フレームオブジェクトは実行フレーム (execution frame) を表します。
実行フレームはトレースバックオブジェクト内に出現します。

(3) トレースバック (traceback) オブジェクト

- ・トレースバックオブジェクトは例外のスタックトレースを表します。
トレースバックオブジェクトは例外が発生した際に生成されます。
例外ハンドラの検索が実行スタックを戻っていく際、戻ったレベル毎に、
トレースバックオブジェクトが現在のトレースバックの前に挿入されます。

(4) スライス (slice) オブジェクト

- ・スライスオブジェクトは、 `__getitem__()` メソッドのためのスライスを表します。

(5) 静的メソッド (static method) オブジェクト

- ・静的メソッドオブジェクトは他の何らかのオブジェクト、通常はユーザ定義メソッドオブジェクトを包むラップです。
- ・静的メソッドをクラスやクラスインスタンスから取得すると、実際に返されるオブジェクトはラップされたオブジェクトになります。
- ・静的オブジェクトは組み込みコンストラクタ `staticmethod()` で生成されます。

(6) クラスメソッドオブジェクト

- ・クラスメソッドオブジェクトは、静的メソッドオブジェクトに似て、別のオブジェクトを包むラップであり、そのオブジェクトをクラスやクラスインスタンスから取り出します。
- ・クラスメソッドオブジェクトは組み込みのコンストラクタ `classmethod()` で生成されます。

(9.3) 変数のスコープ

- ・関数の内部で生成したオブジェクトは、基本的にはその関数の「ローカル変数（局所変数）」であり、その関数の実行が終了した後は消滅します。
- ・関数の外部で生成された「大域変数」を関数内部で更新するには、当該関数内で大域変数の使用を宣言します。具体的には関数定義の内部で、「global 大域変数の名前」と記述します。

(構文) 大域変数の名前1 = 大域変数の値

```
def 関数名( 仮引数のリスト ) :  
    global 大域変数の名前1  
    . . . 処理 . . .  
    return 戻り値のリスト
```

(実装例) # 変数のスコープのテスト

```
gval = ' (大域変数 gval の初期値)' #大域変数  
  
# 関数内部で、大域変数を使用する例  
def func1():  
    global gval # 大域変数であることの宣言  
    print(' func1 の内部では:', gval )  
    gval = ' func1 が書き換えた大域変数 gval の値です。'  
  
# 関数内部で、大域変数と同名の局所変数を使用する例  
def func2():  
    gval = ' func2 の局所変数 gval の値です。'  
    print(gval)  
  
# メインルーチン  
print('【大域変数 gval の値】')  
print(' func1 呼び出し前:', gval)  
func1()  
print(' func1 呼び出し後:', gval)  
  
func2()  
print(' func2 呼び出し後:', gval)
```

(実行結果) 【大域変数 gval の値】

```
func1 呼び出し前: (大域変数 gval の初期値)  
func1 の内部では: (大域変数 gval の初期値)  
func1 呼び出し後: func1 が書き換えた大域変数 gval の値です。  
func2 の局所変数 gval の値です。  
func2 呼び出し後: func1 が書き換えた大域変数 gval の値です。
```

(9.4) 配列（リスト型）

- ・ 変更可能なシーケンス型の一つが「リスト型（list）」です。
（リスト型は、作成した後で要素を変更したり削除することができます。）
- ・ リスト型のデータは、個々の要素を表現する式をカンマで区切り、全体を角括弧（かっこ） [] で囲んで構成します。
- ・ 「リスト型（list）」は使用頻度が高いので、実装例を掲載します。

```
(実装例) #*****/
# リスト03-(09)-1_変更可能なシーケンス型「リスト型（list）」
#*****/

#(1) 配列の宣言と初期化
#(1.1) 宣言のみ
listName = []

#(1.2) 宣言と同時に要素も追加（初期化）
listName = ["00", "01", "02"]
print("(1.2) 宣言と同時に要素も追加（初期化）：{0}".format(listName))

#(1.3) 宣言と同時に要素も追加（同一の値で個数指定で初期化）
listName = ["00"] * 2
print("(1.3) 宣言と同時に要素も追加（同一の値で個数指定で初期化）：{0}".format(listName))

#(1.4) 宣言と同時に要素も追加（同一の配列で個数指定で初期化）
listName = ["00", "01", "02"] * 2
print("(1.4) 宣言と同時に要素も追加（同一の配列で個数指定で初期化）：{0}".format(listName))

#(2) 配列の長さ、要素の参照
#(2.1) 要素数（長さ）を取得（len()関数）
listName = ["00", "01", "02"]
print("(2.1) 要素数（長さ）を取得（len()関数）：{0}".format(len(listName)))

#(2.2) 要素の参照（0以上のインデックスで参照⇒最初の要素から参照）
listName = ["00", "01", "02"]
print("(2.2) 要素の参照（0以上のインデックスで参照⇒最初の要素から参照）：listName[0]={0}".format(listName[0]))
print("(2.2) 要素の参照（0以上のインデックスで参照⇒最初の要素から参照）：listName[2]={0}".format(listName[2]))

#(2.3) 要素の参照（マイナスのインデックスで参照⇒最後の要素から参照）
listName = ["00", "01", "02"]
print("(2.3) 要素の参照（マイナスのインデックスで参照⇒最後の要素から参照）：listName[-1]={0}".format(listName[-1]))
print("(2.3) 要素の参照（マイナスのインデックスで参照⇒最後の要素から参照）：listName[-3]={0}".format(listName[-3]))

#(3) 要素の追加
#(3.1) 要素の追加（+演算子による追加）
listName = ["00", "01", "02"]
listName = listName + ["03", "04", "05"]
print("(3.1) 要素の追加（+演算子による追加）：{0}".format(listName))

#(3.2) 要素の追加（appendメソッドで、値指定による追加）
listName = ["00", "01", "02"]
listName.append("30")
print("(3.2) 要素の追加（appendメソッドで、値指定による追加）：{0}".format(listName))

#(3.3) 要素の追加（insertメソッドで、インデックス指定による挿入）
listName = ["00", "01", "02"]
listName.insert(2, "30") # 2 はインデックス（0始まり）
print("(3.3) 要素の追加（insertメソッドで、インデックス指定による挿入）：{0}".format(listName))
```



```

#(4) 配列を結合
#(4.1) 配列を結合(extendメソッドで、配列を結合)
listName = ["00", "01", "02"]
listName2 = ["03", "04", "05"]
listName.extend(listName2)
print("(4.1) 配列を結合(extendメソッドで、配列を結合) : {0}".format(listName))

#(4.2) 配列を結合(+演算子で、配列を結合)
listName1 = ["00", "01", "02"]
listName2 = ["03", "04", "05"]
listName = listName1 + listName2
print("(4.2) 配列を結合(+演算子で、配列を結合) : {0}".format(listName))

#(5) 要素の削除
#(5.1) 要素の削除(del文で、インデックス指定による削除)
listName = ["00", "01", "02", "03"]
del listName[2] # 2 はインデックス (0始まり)
print("(5.1) 要素の削除(del文で、インデックス指定による削除) : {0}".format(listName))

#(5.2) 要素の削除(popメソッドで、インデックス指定による削除)
listName = ["00", "01", "02", "03"]
listName.pop(2) # 2 はインデックス (0始まり)
print("(5.2) 要素の削除(popメソッドで、インデックス指定による削除) : {0}".format(listName))

#(5.3) 要素の削除(removeメソッドで、値指定による削除)
listName = ["00", "01", "02", "03"]
listName.remove("02") # "02" は値
print("(5.3) 要素の削除(removeメソッドで、値指定による削除) : {0}".format(listName))

#(6) range型との連携
#(6.1) range からリストを生成したい場合はlist()関数を使用
listName = list(range(3))
print("(6.1) range からリストを生成したい場合はlist()関数を使用 : list(range(3))={0}".format(list(range(3))))
print("(6.1) range からリストを生成したい場合はlist()関数を使用 : list(range(3, 10))={0}".format(list(range(3, 10))))

#(7) 要素の検索
#(7.1) 要素の検索(in演算子で、値指定による検索 : True / False を返す)
listName = ["00", "01", "02", "03"]
print("(7.1) 要素の検索(in演算子で、値指定による検索 : True / False を返す) : {0}".format("02" in listName)) # "02" は値
print("(7.1) 要素の検索(in演算子で、値指定による検索 : True / False を返す) : {0}".format("04" in listName)) # "04" は値

#(7.2) 要素の検索(indexメソッドで、値指定による検索)
listName = ["00", "01", "02", "03"]
print("(7.2) 要素の検索(indexメソッドで、値指定による検索) : {0}".format(listName.index("02"))) # "02" は値
print(listName.index("04"))
#⇒ ValueError: '04' is not in list

```

(実行結果)

```
(1. 2) 宣言と同時に要素も追加(初期化) : ['00', '01', '02']
(1. 3) 宣言と同時に要素も追加(同一の値で個数指定で初期化) : ['00', '00']
(1. 4) 宣言と同時に要素も追加(同一の配列で個数指定で初期化) : ['00', '01', '02', '00', '01', '02']
(2. 1) 要素数(長さ)を取得(len()関数) : 3
(2. 2) 要素の参照(0以上のインデックスで参照⇒最初の要素から参照) : listName[0]=00
(2. 2) 要素の参照(0以上のインデックスで参照⇒最初の要素から参照) : listName[2]=02
(2. 3) 要素の参照(マイナスのインデックスで参照⇒最後の要素から参照) : listName[-1]=02
(2. 3) 要素の参照(マイナスのインデックスで参照⇒最後の要素から参照) : listName[-3]=00
(3. 1) 要素の追加(+演算子による追加) : ['00', '01', '02', '03', '04', '05']
(3. 2) 要素の追加(appendメソッドで、値指定による追加) : ['00', '01', '02', '30']
(3. 3) 要素の追加(insertメソッドで、インデックス指定による挿入) : ['00', '01', '30', '02']
(4. 1) 配列を結合(extendメソッドで、配列を結合) : ['00', '01', '02', '03', '04', '05']
(4. 2) 配列を結合(+演算子で、配列を結合) : ['00', '01', '02', '03', '04', '05']
(5. 1) 要素の削除(del文で、インデックス指定による削除) : ['00', '01', '03']
(5. 2) 要素の削除(popメソッドで、インデックス指定による削除) : ['00', '01', '03']
(5. 3) 要素の削除(removeメソッドで、値指定による削除) : ['00', '01', '03']
(6. 1) range からリストを生成したい場合はlist()関数を使用 : list(range(3))=[0, 1, 2]
(6. 1) range からリストを生成したい場合はlist()関数を使用 : list(range(3, 10))=[3, 4, 5, 6, 7, 8, 9]
(7. 1) 要素の検索(in演算子で、値指定による検索 : True / False を返す) : True
(7. 1) 要素の検索(in演算子で、値指定による検索 : True / False を返す) : False
(7. 2) 要素の検索(indexメソッドで、値指定による検索) : 2
```

ValueError

Traceback (most recent call last)

<ipython-input-8-146847eddf89> in <module>

```
94 listName = ["00","01","02","03"]
```

```
95 print("(7.2) 要素の検索(indexメソッドで、値指定による検索) : {0}".format(listName.index("02"))) # "02" は値
```

→ 96 print(listName.index("04"))

```
97 #⇒ ValueError: '04' is not in list
```

ValueError: '04' is not in list

【参照・引用】

参照⇒ <https://udemy.benesse.co.jp/development/web/python-list.html>

参照⇒ <https://note.nkmk.me/python-list-initialize/>

(9.5) 配列 (タプル型)

- ・ 変更不能なシーケンス型の一つが「タプル型 (tuple)」です。
(タプル型は、作成した後で要素を変更したり削除することができません。)
- ・ タプル型のデータは、個々の要素を表現する式をカンマで区切り、全体を丸括弧(まるかっこ) () で囲んで構成します。
- ・ 「タプル型 (tuple)」は使用頻度が高いので、実装例を掲載します。

```
(実装例) #*****/
# リスト03-(09)-2_変更不能なシーケンス型「タプル型 (tuple)」
#*****/

#(1) 値を指定してタプルを作成
#(1.1) 宣言のみ
tupleName = ()
print("(1.1) 宣言のみ : {0}".format(tupleName))

#(1.2) 宣言と同時に要素も登録
tupleName = ("00", "01", "02")
print("(1.2) 宣言と同時に要素も登録 : {0}".format(tupleName))

#(1.3) 宣言と同時に要素も登録(同一の値で個数指定で登録)
tupleName = ("00",) * 3
print("(1.3) 宣言と同時に要素も登録(同一の値で個数指定で初期化) : {0}".format(tupleName))

#(1.4) 宣言と同時に要素も登録(同一のタプルで個数指定で登録)
tupleName = ("00", "01", "02") * 2
print("(1.4) 宣言と同時に要素も登録(同一のタプルで個数指定で登録) : {0}".format(tupleName))

#(1.5) 変数に代入された値を要素として指定)
val00 = "00"
val01 = "01"
val02 = "02"
tupleName = (val00, val01, val02)
print("(1.5) 変数に代入された値を要素として指定) : {0}".format(tupleName))

#(2) 配列の長さ、要素の参照
#(2.1) 要素数(長さ)を取得(len()関数)
tupleName = ("00", "01", "02")
print("(2.1) 要素数(長さ)を取得(len()関数) : {0}".format(len(tupleName)))

#(2.2) 要素の参照(0以上のインデックスで参照⇒最初の要素から参照)
tupleName = ("00", "01", "02")
print("(2.2) 要素の参照(0以上のインデックスで参照⇒最初の要素から参照) : tupleName[0]={0}".format(tupleName[0]))
print("(2.2) 要素の参照(0以上のインデックスで参照⇒最初の要素から参照) : tupleName[2]={0}".format(tupleName[2]))

#(2.3) 要素の参照(マイナスのインデックスで参照⇒最後の要素から参照)
tupleName = ("00", "01", "02")
print("(2.3) 要素の参照(マイナスのインデックスで参照⇒最後の要素から参照) : tupleName[-1]={0}".format(tupleName[-1]))
print("(2.3) 要素の参照(マイナスのインデックスで参照⇒最後の要素から参照) : tupleName[-3]={0}".format(tupleName[-3]))
```

```

#(2.4) スライス機能を使用して参照
tupleName = ("00", "01", "02", "03", "04")
print("(2.4) スライス機能を使用して参照 : tupleName[1:3]={0}".format(tupleName[1:3]))
print("(2.4) スライス機能を使用して参照 : tupleName[ :3]={0}".format(tupleName[ :3]))
print("(2.4) スライス機能を使用して参照 : tupleName[1: ]={0}".format(tupleName[1: ]))
print("(2.4) スライス機能を使用して参照 : tupleName[ : ]={0}".format(tupleName[ : ]))

#(3) 配列を結合
#(3.1) 配列を結合(+演算子で、配列を結合)
tupleName1 = ("00", "01", "02")
tupleName2 = ("03", "04", "05")
tupleName = tupleName1 + tupleName2
print("(3.1) 配列を結合(+演算子で、配列を結合) : {0}".format(tupleName))

#(4) range型との連携
#(4.1) range からリストを生成したい場合はtuple()関数を使用
tupleName = tuple(range(3))
print("(4.1) range からリストを生成したい場合はtuple()関数を使用 : tuple(range(3))={0}".format(tuple(range(3))))
print("(4.1) range からリストを生成したい場合はtuple()関数を使用 : tuple(range(3, 10))={0}".format(tuple(range(3, 10))))

#(5) 要素の検索
#(5.1) 要素の検索(in演算子で、値指定による検索 : True / False を返す)
tupleName = ("00", "01", "02", "03")
print("(5.1) 要素の検索(in演算子で、値指定による検索 : True / False を返す) : {0}".format("02" in tupleName)) # "02" は値
print("(5.1) 要素の検索(in演算子で、値指定による検索 : True / False を返す) : {0}".format("04" in tupleName)) # "04" は値

#(5.2) 要素の検索(indexメソッドで、値指定による検索)
tupleName = ("00", "01", "02", "03")
print("(5.2) 要素の検索(indexメソッドで、値指定による検索) : {0}".format(tupleName.index("02"))) # "02" は値
print(tupleName.index("04"))
#⇒ ValueError: '04' is not in tuple

```

(実行結果)

```
(1.1) 宣言のみ : ()
(1.2) 宣言と同時に要素も登録 : ('00', '01', '02')
(1.3) 宣言と同時に要素も登録(同一の値で個数指定で初期化) : ('00', '00', '00')
(1.4) 宣言と同時に要素も登録(同一のタプルで個数指定で登録) : ('00', '01', '02', '00', '01', '02')
(1.5) 変数に代入された値を要素として指定 : ('00', '01', '02')
(2.1) 要素数(長さ)を取得(len()関数) : 3
(2.2) 要素の参照(0以上のインデックスで参照⇒最初の要素から参照) : tupleName[0]=00
(2.2) 要素の参照(0以上のインデックスで参照⇒最初の要素から参照) : tupleName[2]=02
(2.3) 要素の参照(マイナスのインデックスで参照⇒最後の要素から参照) : tupleName[-1]=02
(2.3) 要素の参照(マイナスのインデックスで参照⇒最後の要素から参照) : tupleName[-3]=00
(2.4) スライス機能を使用して参照 : tupleName[1:3]=('01', '02')
(2.4) スライス機能を使用して参照 : tupleName[:3]=('00', '01', '02')
(2.4) スライス機能を使用して参照 : tupleName[1:]=('01', '02', '03', '04')
(2.4) スライス機能を使用して参照 : tupleName[:]=('00', '01', '02', '03', '04')
(3.1) 配列を結合(+演算子で、配列を結合) : ('00', '01', '02', '03', '04', '05')
(4.1) range からリストを生成したい場合はtuple()関数を使用 : tuple(range(3))=(0, 1, 2)
(4.1) range からリストを生成したい場合はtuple()関数を使用 : tuple(range(3, 10))=(3, 4, 5, 6, 7, 8, 9)
(5.1) 要素の検索(in演算子で、値指定による検索 : True / False を返す) : True
(5.1) 要素の検索(in演算子で、値指定による検索 : True / False を返す) : False
(5.2) 要素の検索(indexメソッドで、値指定による検索) : 2
```

ValueError

Traceback (most recent call last)

<ipython-input-9-595409a45e49> in <module>

```
72 tupleName = ("00","01","02","03")
```

```
73 print("(5.2) 要素の検索(indexメソッドで、値指定による検索) : {0}".format(tupleName.index("02"))) # "02" は値
```

—> 74 print(tupleName.index("04"))

```
75 #⇒ ValueError: '04' is not in tuple
```

ValueError: tuple.index(x): x not in tuple

(9.6) 辞書型

- ・他の高級言語と同様に、Python にも「辞書型 (dictionary)」が組み込まれています。
辞書は キー(key) でインデクス化されていて、キーに紐づく値(value) を参照・登録します。
キー(key) にできるのは、文字列、数値等です。
値(value) にできるのは、任意のオブジェクトです。
- ・辞書型のデータは、個々の要素は「キー : 値」で記述し、各要素をカンマで区切り、全体を波括弧(なみかっこ) { } で囲んで構成します。
- ・「辞書型」は使用頻度が高いので、実装例を掲載します。

```
(実装例) #*****/  
# リスト03-(09)-3_辞書型 (dictionary)  
#*****/  
  
#(1) 辞書の作成と初期化  
#(1.1) 空の辞書を作成  
dictName = {}  
print("(1.1) 空の辞書を作成")  
print(dictName)  
  
#(1.2) 作成と同時に登録(カンマで区切られた key: value のペアを代入演算子で初期登録)  
dictName = {"key00":"value00","key01":"value01","key02":"value02","key00":"value00","key02":"value02"}  
print("(1.2) 作成と同時に登録(カンマで区切られた key: value のペアを指定)")  
print(dictName)  
  
#(1.3) dict() コンストラクタによる作成 その1  
dictName = dict([("key00","value00"), ("key01","value01"), ("key02","value02"), ("key00","value00"), ("key02","value02")])  
print("(1.3) dict() コンストラクタによる作成 その1")  
print(dictName)  
  
#(1.4) dict() コンストラクタによる作成 その2  
dictName = dict(key00="value00", key01="value01", key02="value02")  
print("(1.3) dict() コンストラクタによる作成 その2")  
print(dictName)  
  
#(2) 要素の追加  
#(2.1) 要素の追加(代入演算子による追加)  
dictName = {"key00":"value00","key01":"value01","key02":"value02"}  
dictName["key03"] = "value03"  
print("(2.1) 要素の追加(代入演算子による追加)")  
print(dictName)  
  
#(3) 要素の削除  
#(3.1) 要素の削除(del文で、キー指定による削除)  
dictName = {"key00":"value00","key01":"value01","key02":"value02"}  
del dictName["key01"]  
print("(3.1) 要素の削除(del文で、キー指定による削除)")  
print(dictName)
```

```

#(4) 要素の検索
#(4.1) 要素の検索(in演算子で、キー指定による検索 : True / False を返す)
dictName = {"key00":"value00","key01":"value01","key02":"value02"}
print("(4.1) 要素の検索(in演算子で、キー指定による検索 : True / False を返す)")
print("  {0} : {1}".format("key01", "key01" in dictName))
print("  {0} : {1}".format("key04", "key04" in dictName))

#(4.2) 要素の検索(not in演算子で、キー指定による検索 : True / False を返す)
dictName = {"key00":"value00","key01":"value01","key02":"value02"}
print("(4.2) 要素の検索(not in演算子で、キー指定による検索 : True / False を返す)")
print("  {0} : {1}".format("key01", "key01" not in dictName))
print("  {0} : {1}".format("key04", "key04" not in dictName))

#(4.3) 要素の検索(list文によるキーの表示)
dictName = {"key00":"value00","key01":"value01","key02":"value02"}
print("(4.3) 要素の検索(list文によるキーの表示)")
print(list(dictName))

#(4.4) 要素の検索(for文による検索)
print("(4.4) 要素の検索(for文による検索)")
dictName = {"key00":"value00","key01":"value01","key02":"value02"}
for key, val in dictName.items():
    print("  {0} : {1}".format(key, val))

```

(実行結果)

```

(1.1) 空の辞書を作成
[]
(1.2) 作成と同時に登録(カンマで区切られた key: value のペアを指定)
{'key00': 'value00', 'key01': 'value01', 'key02': 'value02'}
(1.3) dict() コンストラクタによる作成 その1
{'key00': 'value00', 'key01': 'value01', 'key02': 'value02'}
(1.3) dict() コンストラクタによる作成 その2
{'key00': 'value00', 'key01': 'value01', 'key02': 'value02'}
(2.1) 要素の追加(代入演算子による追加)
{'key00': 'value00', 'key01': 'value01', 'key02': 'value02', 'key03': 'value03'}
(3.1) 要素の削除(del文で、キー指定による削除)
{'key00': 'value00', 'key02': 'value02'}
(4.1) 要素の検索(in演算子で、キー指定による検索 : True / False を返す)
key01 : True
key04 : False
(4.2) 要素の検索(not in演算子で、キー指定による検索 : True / False を返す)
key01 : False
key04 : True
(4.3) 要素の検索(list文によるキーの表示)
['key00', 'key01', 'key02']
(4.4) 要素の検索(for文による検索)
key00 : value00
key01 : value01
key02 : value02

```

(9.7) 配列 (Numpy.ndarray)

- ・ライブラリ「Numpy」には「ndarray」という「Numpy」で使われる配列があります。
「ndarray」は多次元配列を扱うためのクラスです。

- (1) 1次元の配列を作るためには、
リスト型配列として作成します。
- (2) 2次元の行列を作るためには、
行ベクトルを一つのリスト型配列で表し、
行ベクトル数分カンマで区切って、全体のリスト型配列を作成し、
2次元の行列とします。

- ・「Numpy.ndarray」は使用頻度が高いので、実装例を掲載します。

(実装例)

```
#####/  
# リスト03-(09)-4_Numpyの配列 (Numpy.ndarray)  
#####/  
import numpy as np  
  
#(1) 1次元の配列  
npArray1 = np.array(["00", "01", "02"])  
print("1次元の配列 : %n{0}%n".format(npArray1))  
  
#(2) 2次元の配列  
npArray2 = np.array([["00", "01", "02"], ["10", "11", "12"]])  
print("2次元の配列 : %n{0}%n".format(npArray2))
```

(実行結果)

```
1次元の配列 :  
['00' '01' '02']  
  
2次元の配列 :  
[['00' '01' '02']  
 ['10' '11' '12']]
```

【参照・引用】

参照⇒ <https://docs.python.org/ja/3/tutorial/datastructures.html>

(10) 単純文

Python プログラムの単純文について、下記サイト

⇒ https://docs.python.org/ja/3.6/reference/simple_stmts.html

などを参照・引用しながら、その要約を記します。

(10.1) Python の単純文

- ・単純文 (simple statement) とは、単一の論理行内に収められる文です。
- ・単一の行内には、複数の単純文をセミコロンで区切って入れることができます。
- ・以下に Python の単純文一覧を示します。

単純文名	説明
式文 (expression statement)	・式文は式のリスト (単一の式のこともあります) を値評価します。
代入文 (assignment statement)	・代入文は式のリスト (これは単一の式でも、カンマで区切られた式リストでもよい) を評価し、得られた単一の結果オブジェクトをターゲットのリストに対して左から右へと代入します。 (実装例) <pre>x = x + 1</pre>
累算代入文 (augmented assignment statement)	・累算代入文は、ターゲット (通常の代入文と違って、アンパックは起こりません) と式リストを評価し、それら二つの被演算子間で特定の累算代入型の二項演算を行い、結果を元のターゲットに代入します。 (実装例) <pre>x += 1</pre>
注釈付き代入文 (annotated assignment statements)	・代入先として単純名を使うと、クラススコープもしくはモジュールスコープの場合、注釈は評価され、クラスもしくはモジュールの特殊属性 <code>__annotations__</code> に格納されます。 ・代入先として式を使うと、クラススコープもしくはモジュールスコープの場合、注釈は評価されますが、格納されません。
assert 文	・assert 文は、プログラム内にデバッグ用アサーションを仕掛けるための方法です。
pass 文	・pass 文はヌル操作です。pass が実行されても何も起きません。 ・pass 文は構文法的には文が必要だが、コードとしては何も実行したくない場合に用います。
del 文	・del 文で、ターゲットリストに対し各々のターゲットを左から右へと順に再帰的に削除します。
return 文	・return 文で、式リスト (または None) を戻り値として、現在の関数呼び出しから抜け出します。 (実装例) <pre>def f3(x, y): ans = np.exp(-x**2 - y**2) return ans, x, y</pre>
yield 文	・関数定義内で yield 文を使用することで、その定義は通常の関数でなくジェネレータ関数になります。そのような関数は、呼び出されたとき、イテレータオブジェクトを返します。 (実装例) <pre>def func(): yield 'sample-1' yield 'sample-2' yield 'sample-3' for ret in func(): print ret</pre> (実行結果) <pre>sample-1 sample-2 sample-3</pre>

raise 文	<ul style="list-style-type: none"> • raise 文で式を伴わない場合、現在のスコープで最終的に有効になっている例外を再送出します。 • raise 文で式を伴う場合、最初の式を、例外オブジェクトとして送出します。 <p>(実装例)</p> <pre>try: print(1 / 0) except: raise RuntimeError("Something bad happened") from None</pre>
break 文	<ul style="list-style-type: none"> • break 文は、文を囲う最も内側のループを終了させ、ループにオプションの else 節がある場合にはそれをスキップします。
continue 文	<ul style="list-style-type: none"> • continue 文は、文を囲う最も内側のループの次の周期に処理を継続します。
import 文	<ul style="list-style-type: none"> • import 文で、外部のモジュールを内部に取り込みます。 • import 文は 2 つのステップで実行されます: <ol style="list-style-type: none"> (1) モジュールを見付け出し、必要であればロードし初期化する。 (2) import 文が表れるスコープのローカル名前空間で名前を定義する。 <p>(実装例)</p> <pre>import matplotlib.pyplot as plt from mpl_toolkits.mplot3d import Axes3D</pre>
future 文 (future statement)	<ul style="list-style-type: none"> • future 文 は、将来の特定の新たな機能が標準化された Python のリリースで利用可能になるような構文や意味付けを使って、特定のモジュールをコンパイルさせるための、コンパイラに対する指示句 (directive) です。
global 文	<ul style="list-style-type: none"> • global 文は、列挙した識別子をグローバル変数として解釈するよう指定することを意味します。 • global 文で列挙する名前は、同じコードブロック中で、プログラムテキスト上 global 文より前に使ってはなりません。
nonlocal 文	<ul style="list-style-type: none"> • nonlocal 文は、列挙された識別子がグローバルを除く一つ外側のスコープで先に束縛された変数を参照するようにします。 • nonlocal 文で列挙された名前は、ローカルスコープですでに存在する束縛と衝突してはなりません。

(11) 制御構文

Python プログラムの制御文について、下記サイト

⇒ https://docs.python.org/ja/3.6/reference/compound_stmts.html

⇒ <https://docs.python.org/ja/3.6/reference/datamodel.html#context-managers>

などを参照・引用しながら、その要約を記します。

(11.1) if 文

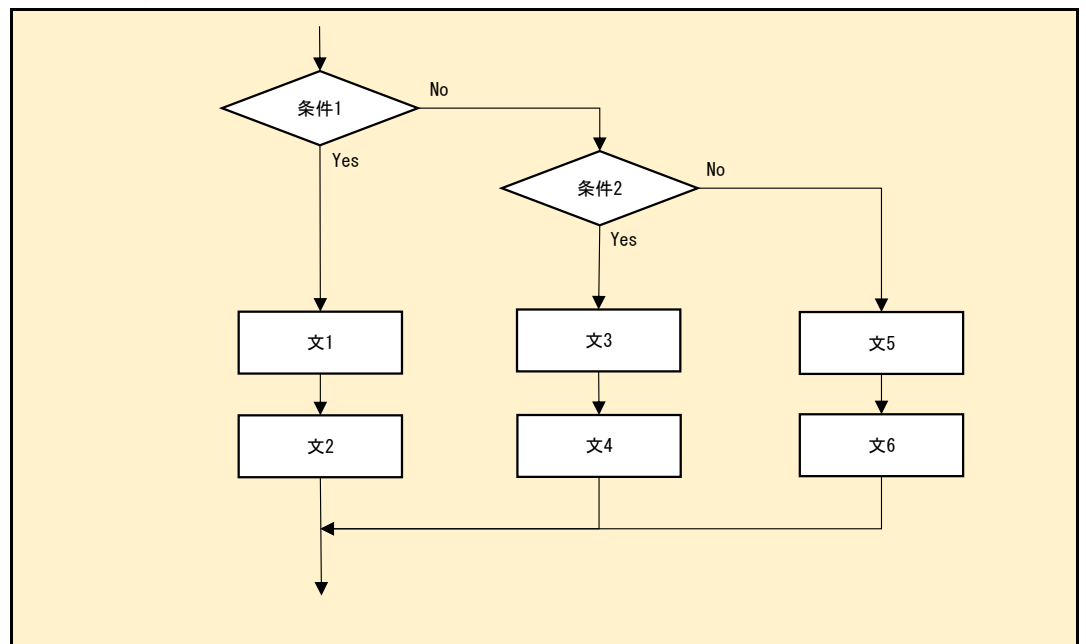
if 文は、条件分岐を実行するために使われます。

(構文)

```
if_stmt ::= "if" expression ":" suite
          ("elif" expression ":" suite)*
          ["else" ":" suite]
```

(実装パターンとそのフローチャート)

```
if 条件1 :
    文1
    文2
elif 条件2 :
    文3
    文4
else:
    文5
    文6
```



(11.2) for 文

- ・ for 文は、シーケンス(文字列、タプルまたはリスト)や、その他の反復可能なオブジェクト(iterable object)内の要素に渡って反復処理を行うために使われます。

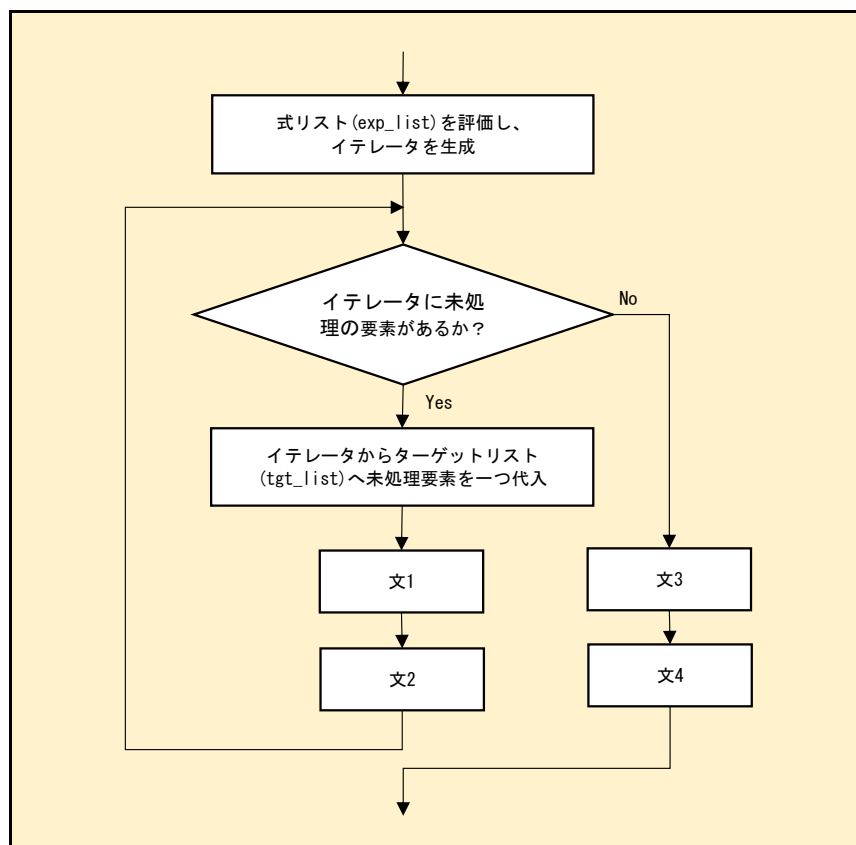
(構文)

```
for_stmt ::= "for" target_list "in" expression_list ":" suite  
          ["else" ":" suite]
```

- ・ 式リスト(expression_list)は一度だけ評価され、これはイテラブルオブジェクトを与えなければなりません。
 - ・ 式リストの結果に対するイテレータが生成されます。その後、イテレータが与えるそれぞれの要素に対してイテレータに返された順に一度ずつ、スイート(suite)が実行されます。
 - ・ それぞれの要素は通常の代入規則でターゲットリスト(target_list)に代入され、その後スイート(suite)が実行されます。
- 全ての要素を使い切ったとき、else 節があればそれが実行され、ループは終了します。

(実装パターンとそのフローチャート)

```
for tgt_list in exp_list :  
    文1  
    文2  
else:  
    文3  
    文4
```



(11.3) while 文

- ・ while 文は、式の値が真である間、実行を繰り返すために使われます。

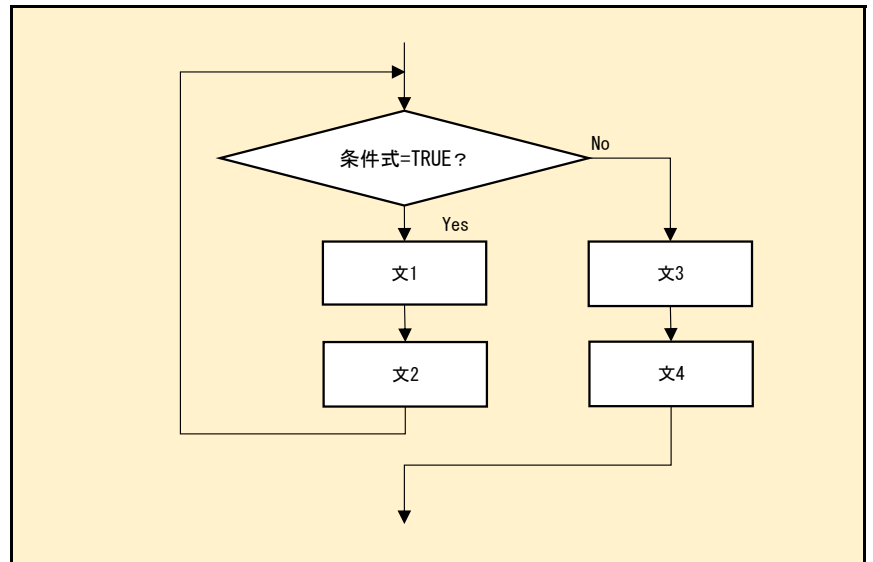
(構文)

```
while_stmt ::= "while" expression ":" suite  
            ["else" ":" suite]
```

- ・ while 文は式(expression)を繰り返し真偽評価し、真であれば最初のスイート(suite)を実行します。
- ・ 式が偽であれば、 else 節がある場合にはそれ(suite)を実行し、ループを終了します。

(実装パターンとそのフローチャート)

```
while 条件式 :  
    文1  
    文2  
else:  
    文3  
    文4
```



(11.4) with 文

- ・ with 文は、ブロックの実行を、コンテキストマネージャによって定義されたメソッドでラップするために使われます。
- ・ コンテキストマネージャ(context manager)は、コードブロックを実行するために必要な入り口および出口の処理を扱います。
 - (1) 「__enter__」メソッドが、コンテキストマネージャの入り口で実行される処理です。
 - (2) 「__exit__」メソッドが、コンテキストマネージャの出口で実行される処理です。

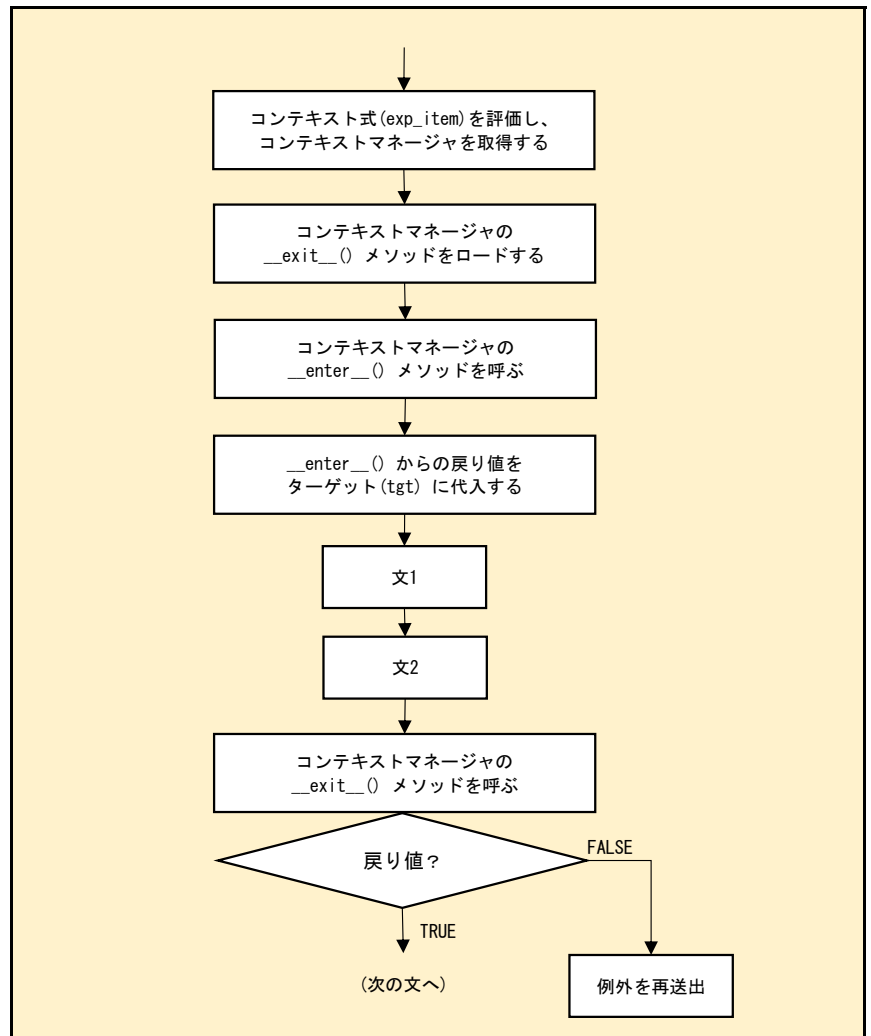
(構文)

```
with_stmt ::= "with" with_item ("," with_item)* ":" suite
with_item ::= expression ["as" target]
```

- ・ 一つの“要素”を持つ with 文の実行は以下のように進行します：
 - (1) コンテキスト式 (with_item) を評価することで、コンテキストマネージャを取得します。
 - (2) コンテキストマネージャの __exit__() メソッドが、後で使うためにロードされます。
 - (3) コンテキストマネージャの __enter__() メソッドが呼ばれます。
 - (4) with 文にターゲット(target)が含まれていたなら、それに __enter__() からの戻り値が代入されます。
 - (5) スイート(suite)が実行されます。
 - (6) コンテキストマネージャの __exit__() メソッドが呼ばれます。
 - (7) スイートが例外により終了され、__exit__() メソッドからの戻り値が偽 (false) ならば、例外が再送出されます。この戻り値が真 (true) ならば例外は抑制され、実行は with 文の次の文から続きます。もしそのスイートが例外でない何らかの理由で終了した場合、その __exit__() からの戻り値は無視されて、実行は発生した終了の種類に応じた通常の位置から継続します。

(実装パターンとそのフローチャート)

```
with exp_item as tgt :
    文1
    文2
```



(11.5) try 文

- ・ try 文は、ひとまとめの文に対して、例外処理とクリーンアップコードを指定します。

(構文)

```
try_stmt ::= try1_stmt | try2_stmt
try1_stmt ::= "try" ":" suite
              ("except" [expression ["as" identifier]] ":" suite)+
              ["else" ":" suite]
              ["finally" ":" suite]
try2_stmt ::= "try" ":" suite
              "finally" ":" suite
```

- ・ except 節は一つ以上の例外ハンドラを指定します。
- ・ try 節内で例外が起きなければ、どの例外ハンドラも実行されません。
- ・ try スイート(suite)内で例外が発生すると、例外ハンドラの検索が開始されます。
この検索では、except 節を逐次、発生した例外に対応するまで調べます。
式を伴わない except 節を使うなら、最後に書かなければならず、これは全ての例外に対応します。
式を伴う except 節に対しては、その式が評価され、結果のオブジェクトが例外と“互換である (compatible)”場合にその節に対応します。
ある例外に対してオブジェクトが互換であるのは、それが例外オブジェクトのクラスかベースクラスの場合、または例外と互換である要素が入ったタプルである場合です。
例外がどの except 節にも合致しなかった場合、現在のコードを囲うさらに外側、そして呼び出しスタックへと検索を続けます。
- ・ オプションの else 節は、実行の制御が try 節の末尾に到達した場合に実行されます。
else 節内で起きた例外は、else 節に先行する except 節で処理されることはありません。
- ・ オプションの finally 節には、'後始末 (cleanup)' の対処を指定します。
finally 節は、except 節や else 節を含めて try 節が実行された後で必ず実行されます。
try 節の実行で発生した例外は、finally 節が始まる時に一時的に保存され、finally 節の末尾で再送出されます。但し、finally 節で return 文あるいは break 文を実行した場合は、保存された例外は破棄されます。
- ・ 関数の戻り値は最後に実行された return 文によって決まります。
finally 節は必ず実行されるため、finally 節で実行された return 文は常に最後に実行されます。

(実装例)

```
def foo(a,b):
    val = 9
    try:
        div = a / b
        val = 0
    except Exception as e:
        print('Exception occurred .. {0}'.format(e))
    finally:
        return val

print('foo(1,0)={0}'.format(foo(1,0)))
print('foo(1,2)={0}'.format(foo(1,2)))
```

(実行結果)

```
Exception occurred .. division by zero
foo(1,0)=9
foo(1,2)=0
```

(12) 関数

Python プログラムの関数定義について、下記サイト

⇒ https://docs.python.org/ja/3.6/reference/compound_stmts.html

などを参照・引用しながら、その要約を記します。

(12.1) 関数定義

- ・関数定義は、ユーザ定義関数オブジェクトを定義します。

(構文)

```
funcdef ::= [decorators] "def" funcname "(" [parameter_list] ")"  
        ["->" expression] ":" suite  
decorators ::= decorator+  
decorator ::= "@" dotted_name "(" [argument_list [","]] ")" NEWLINE  
dotted_name ::= identifier ( "." identifier ) *  
parameter_list ::= defparameter ( "," defparameter ) * [ "," [parameter_list_starargs] ]  
                | parameter_list_starargs  
parameter_list_starargs ::= "*" [parameter] ( "," defparameter ) * [ "," ["**" parameter [ "," ] ] ]  
                | "**" parameter [ "," ]  
parameter ::= identifier [ ":" expression ]  
defparameter ::= parameter [ "=" expression ]  
funcname ::= identifier
```

(関数定義の実装パターン)

```
def 関数名( 仮引数のリスト ) :  
    """関数の説明文 (function's documentation string)  
    """  
    . . . 処理 . . .  
    return 戻り値のリスト
```

(実装例)

```
def whats_on_the_telly(penguin=None):  
    if penguin is None:  
        penguin = []  
    penguin.append("property of the zoo")  
    return penguin
```

- ・関数定義内の処理とreturnまでの行は、defよりも右の位置に同一の深さの字下げをして記述します。
- ・関数名の値は、ユーザ定義関数として、別の名前に割り当てることができ、それを関数として使用することもできます。

(12.2) 戻り値

- ・return 文で戻り値を返しますが、return 文を書かない場合、None が返ります。
- ・どんな型でも戻り値にできます。
- ・カンマ「,」で区切るにより、戻り値を複数指定できます。

その場合、呼び出し側は、受け取る変数をカンマ「,」で区切って複数並べるにより、戻り値を複数取得できます。あるいは、tuple 型として一つの変数で受け取ることも出来ます。

(12.3) 引数（仮引数と実引数）

- ・ 仮引数（formal parameter）が「parameter = expression」の書式でデフォルト値を指定することで、関数呼び出し時の実引数（argument）指定は省略可能です。
このデフォルト値は、関数定義時点で評価されるものであり、変数での指定も可能です。
また、デフォルト引数値は関数定義が実行されるときに左から右へ1回だけ評価されます。
- ・ 実引数は「値渡し（call by value）」を使用して渡されます。
- ・ 実引数の指定の仕方には2種類あります。
 - (1) キーワード引数
 - ・ これは、関数呼び出しで識別子（例えば、name =）が前に付いているか、**の前に辞書の値として渡された引数です。
 - (2) 位置引数
 - ・ これは、キーワード引数ではない引数であり、引数リストの先頭に出現するか、*で始まる反復可能要素の要素として渡されます。
 - (3) 位置引数とキーワード引数の混合
 - ・ 位置引数とキーワード引数の混合で指定することもできますが、その場合、以下の条件を満たすことが必要です：
 - ・ 必須の引数がある。
 - ・ 位置引数はキーワード引数に先行する。
 - ・ 同じ引数の重複値指定がない。
 - ・ 不明なキーワード引数がない。
- ・ 「** name」という形式の仮引数の場合、
関数内ではキーワード引数を含む辞書（マッピング型 dict）として参照します。
これに対応する関数呼び出し側では、キーワード引数を含む辞書（マッピング型 dict）として設定した変数を、実引数として渡します。
- ・ 「* name」という形式の仮引数は、仮引数の個数が予め決まっていない場合に指定するもので、関数内ではリストとして参照します。
これに対応する関数呼び出し側では、リストとして設定した変数を、実引数として渡します。
- ・ 仮引数の並びとして、「* name」は「** name」の前になければなりません。
また、「* name」と「** name」の後続の引数は、キーワード引数に限られます。

(実装例)

```
# 関数定義
def oneAsterFunc( *alist ):
    n = len(alist)
    for m in range(n):
        print( alist[m] )
    return n

# 関数呼び出し
alistLen = oneAsterFunc('A', 'b', 'C', 'd')
print('oneAsterFunc の引数の数', alistLen)
```

(結果)

```
A
b
C
d
oneAsterFunc の引数の数 4
```

(12.4) ラムダ式 (lambda expression)

ラムダ式 (ラムダ形式とも呼ばれる) は、匿名関数の作成に使用されます。

(構文)

```
lambda_expr ::= "lambda" [parameter_list]: expression
lambda_expr_nocond ::= "lambda" [parameter_list]: expression_nocond
```

・これは次の関数定義と同義です:

```
def <lambda>(parameter_list):
    return expression
```

(実装例)

```
# ラムダ式の例 (最大値取得)
maxVal = lambda x,y : (x if (x > y) else y)

# 上記ラムダ式の呼び出し
ret = maxVal( 100, 200 )
print('maxVal( 100, 200 ) の戻り値', ret )
ret = maxVal( 200, 100 )
print('maxVal( 200, 100 ) の戻り値', ret )
```

(結果) maxVal(100, 200) の戻り値 200
maxVal(200, 100) の戻り値 200

(12.5) 注釈 (annotation)

- ・「注釈 (annotation)」は関数の「__annotations__」属性に、ディクショナリとして格納されます。
- ・「注釈 (annotation)」には何種類かあります。

(1) パラメータ注釈

- ・これは、パラメータ名の後にコロンで定義され、その後に注釈の値を評価する式が続きます。

(2) 戻り注釈

- ・これは、パラメーターリストとdefステートメントの終了を示すコロンの間のリテラル "->"、続いて式で定義されます。

(実装例)

```
def f(ham: str , eggs: str = 'eggs') -> str:
    return ham + ' and ' + eggs

# Call function f's annotation described above
print("Annotations:", f.__annotations__)
print(f('HAM', 'EGGS'))
```

(結果)

```
Annotations: {'ham': <class 'str'>, 'eggs': <class 'str'>, 'return': <class 'str'>}}
HAM and EGGS
```

(12.6) 関数の呼び出し

- ・関数は、以下の形式で呼び出して実行します。
- ・関数は、その処理結果を戻り値として呼び出し元に返します。

(関数呼び出しの実装パターン)

```
戻り値を受け取る変数のリスト = 関数名( 実引数のリスト )
```

(13) モジュール

Python プログラムのモジュールのインポートについて、下記サイト

⇒ <https://docs.python.org/ja/3/tutorial/modules.html>

⇒ <https://docs.python.org/ja/3/reference/import.html>

などを参照・引用しながら、その要約を記します。

(13.1) モジュール (module)

- ・ Python では定義をファイルに書いておき、他のスクリプトやインタプリタの対話インスタンス上で使う方法があります。このファイルを「モジュール (module)」と呼びます。

(13.1.1) モジュールの定義

- ・ モジュールは Python の定義や文が入ったファイルです。
モジュール名に接尾語 `.py` がついたものが、そのモジュールのファイル名になります。
- ・ 例えば、以下の実装を「`fibonacci.py`」というファイル名で、任意のディレクトリに保存すると、「`fibonacci`」という名前のモジュールが出来ます。

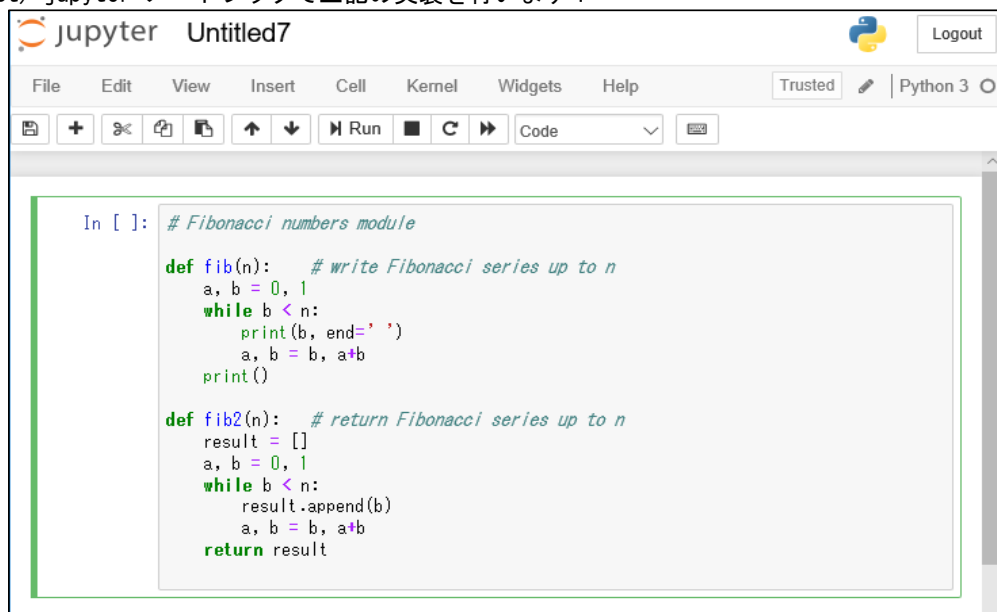
(実装例)

```
#####/
# リスト03-(13)-1_モジュールの定義
# ("fibonacci.py" というファイル名で任意のディレクトリに保存)
#####/
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

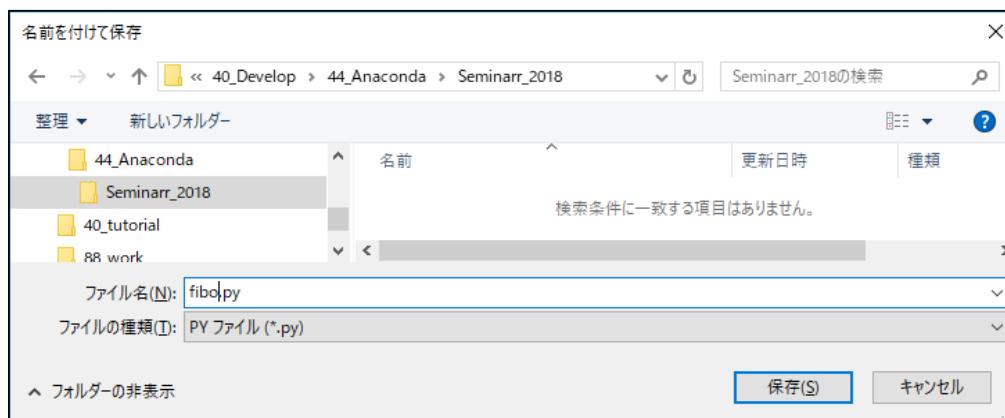
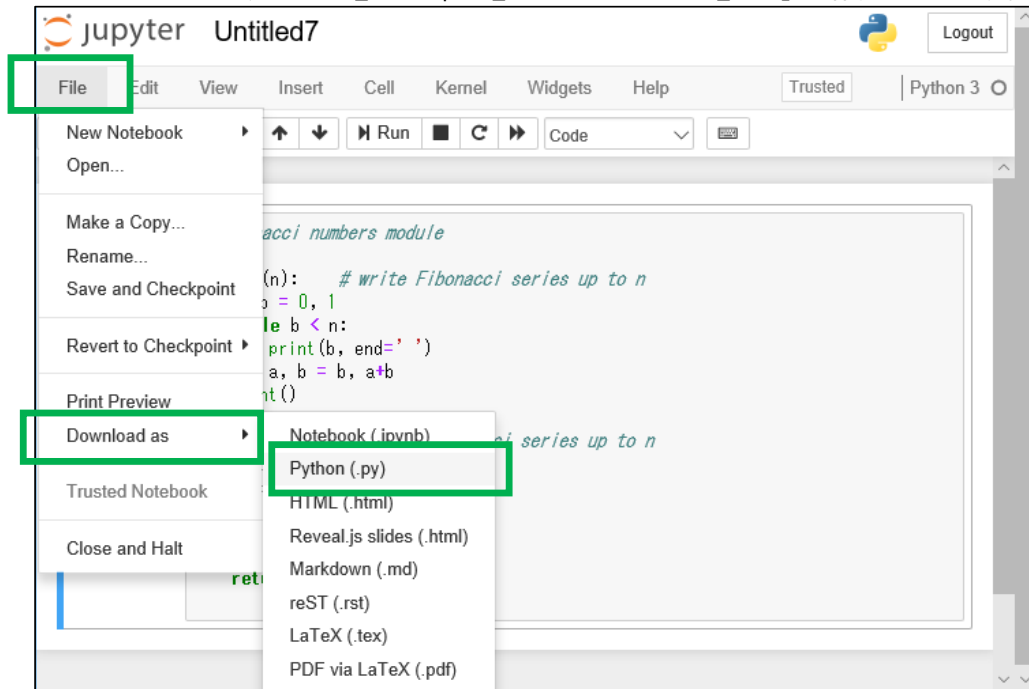
def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

- ・ jupyter ノートブックからだと、以下の様な手順になります：
(1st) jupyter ノートブックで上記の実装を行います：



The screenshot shows a Jupyter Notebook window titled "Untitled7". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations and code execution. The code cell contains the same Python code as shown in the previous block, defining the `fib` and `fib2` functions for calculating Fibonacci numbers. The code is displayed with syntax highlighting.

(2nd) メニューで「File -> Download as -> Python(.py)」で「名前を付けて保存」で「fibonacci.py」というファイル名で、任意のディレクトリに保存します。
(以下の例では、「E:\40_Develop\44_Anaconda\Seminar_2018」へ保存しています。)



(13.1.2) モジュールの取り込み (import)

- ・モジュールにある定義は、他のモジュールに「取り込み (import)」することができます。
- ・「取り込み (import)」する方法にはいくつかあります。

(13.1.2.1) import 文によるモジュールの取り込み

- ・import文によりモジュールを取り込むことができます。

(構文)

```
import_stmt ::= "import" module ["as" identifier] ("," module ["as" identifier])*
              | "from" relative_module "import" identifier ["as" identifier]
              ("," identifier ["as" identifier])*
              | "from" relative_module "import" "(" identifier ["as" identifier]
              ("," identifier ["as" identifier])* [","] ")"
              | "from" module "import" "*"
module       ::= (identifier ".")* identifier
relative_module ::= "."* module | "."+
```

- ・例えば、「(13.1.1) モジュールの定義」で作成した“fibo” という名前のモジュールを参照する場合、
(「モジュール検索パス (sys.path)」に、モジュール保存先のパスを追加したうえで)
import文を実行して取り込みます。
これにより、モジュール名 “fibo” だけが現在のシンボルテーブルに入り、
関数にはモジュール名を使ってアクセスします。

(実装例)

```
#!/usr/bin/env python
# リスト03-(13)-2_import 文によるモジュールの取り込み
# (モジュールを定義した“fibo.py” ファイルのパスを指定してインポート)
#!/usr/bin/env python
import sys
sys.path.append("E:\¥40_Develop¥44_Anaconda¥¥Seminarr_2018")
import fibo

fibo.fib(1000)
fibo.fib2(100)
fibo.__name__
```

(※注意！：「sys.path.append(～)」はその都度、既登録の有無に関わらず、path へ追加されます。)

(実行結果)

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
'fibo'
```

【出典・参考】

⇒ <https://qiita.com/Accent/items/efb57d66309f6c0ee63d>

(13.1.2.2) import 文以外によるモジュールの取り込み

- ・「importlib.import_module()」や組み込みの「__import__()」といった関数を使っても、インポート機構を動かすことができます。
- ・これらについては、以下のサイトを参照してください：
⇒ https://docs.python.org/ja/3.6/library/importlib.html#importlib.import_module
⇒ https://docs.python.org/ja/3.6/library/functions.html#__import__

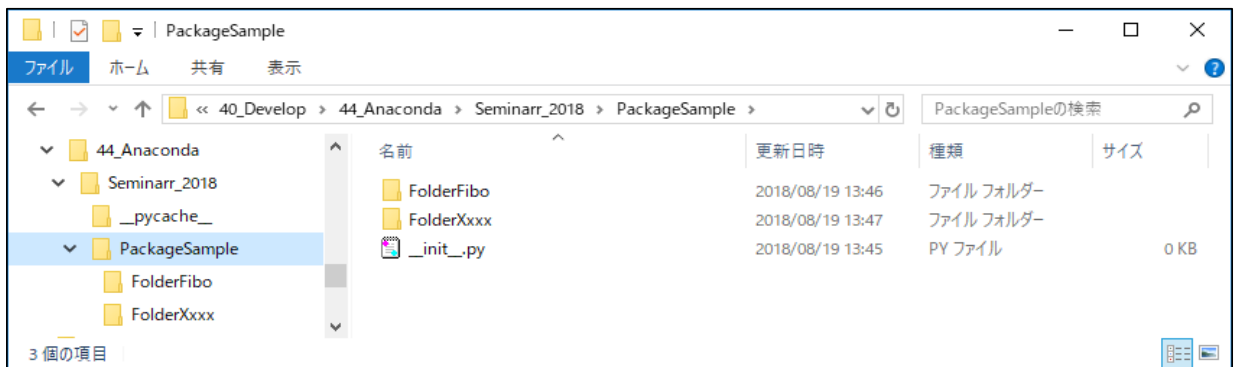
(13.2) パッケージ (package)

- ・ モジュールの組織化を助け、名前階層を提供するために、Python には「パッケージ (package)」という手法があります。
- ・ パッケージ (package) は、Python のモジュール名前空間を “ドット付きモジュール名” を使って構造化する手法です。例えば、モジュール名 A.B は、A というパッケージのサブモジュール B を表します。

(13.2.1) パッケージの定義

- ・ パッケージは、サブモジュールや再帰的にサブパッケージを含むことの出来る module のことで、「__path__」属性を持つ Python オブジェクトです。
- ・ 物理的にはフォルダ構成として、階層的にモジュールのファイル群を配置します。その最上位の階層のフォルダ名がパッケージ名になります。
- ・ あるディレクトリを、パッケージが入ったディレクトリとして Python に扱わせるには、ファイル「__init__.py」が必要です。
- ・ ファイル「__init__.py」はただの空ファイルで構いませんが、「__init__.py」ではパッケージのための初期化コードを実行したり、「__all__」変数を設定してもかまいません。
- ・ 例えば、「(13.1.1) モジュールの定義」で作成した“fibo” という名前のモジュールの配置先を以下の様に “PackageSample” フォルダ以下に構成しなとして、空のファイル「__init__.py」を各フォルダに配置します。これにより、モジュール “fibo” をパッケージ “PackageSample” に含まれるモジュールの一つとしてアクセス出来るようになります。

(実装例)



```
E:\40_Develop\44_Anaconda\Seminarr_2018\PackageSample
| __init__.py
|─ FolderFibo
|   fibo.py
|   __init__.py
|─ FolderXxxx
|   Xxxx.py
|   __init__.py
```

(13.2.2) パッケージ内のモジュールの取り込み

- ・パッケージ内のモジュールの取り込みは、「(13.1.2) モジュールの取り込み (import)」と同様です。違いは、パッケージのパスの階層をドット(.)区切りで指定することが必要になることです。
- ・パッケージのパスの階層をドット(.)区切りで指定したモジュール名を、“as” で名前を簡略にして参照するのが便利です（下の例を参照）。
- ・以下は、「(13.2.1) パッケージの定義」で作成した “PackageSample” という名前のパッケージ内のフォルダ “FolderFibo” 直下にあるモジュール “fibo” を参照する実装例です。

(実装例)

```
import sys
sys.path.append("E:\¥40_Develop¥¥44_Anaconda¥¥Seminarr_2018")
import PackageSample.FolderFibo.fibo as fiboAlias

fiboAlias.fib(1000)
fiboAlias.fib2(100)
fiboAlias.__name__
```

(※注意！：「sys.path.append(～)」はその都度、既登録の有無に関わらず、path へ追加されます。)

(実行結果)

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
'PackageSample.FolderFibo.fibo'
```

(14) オブジェクト指向

Python は、オブジェクト指向言語です。ここでオブジェクト指向を概観し、Pythonでの実装についてみておきます。

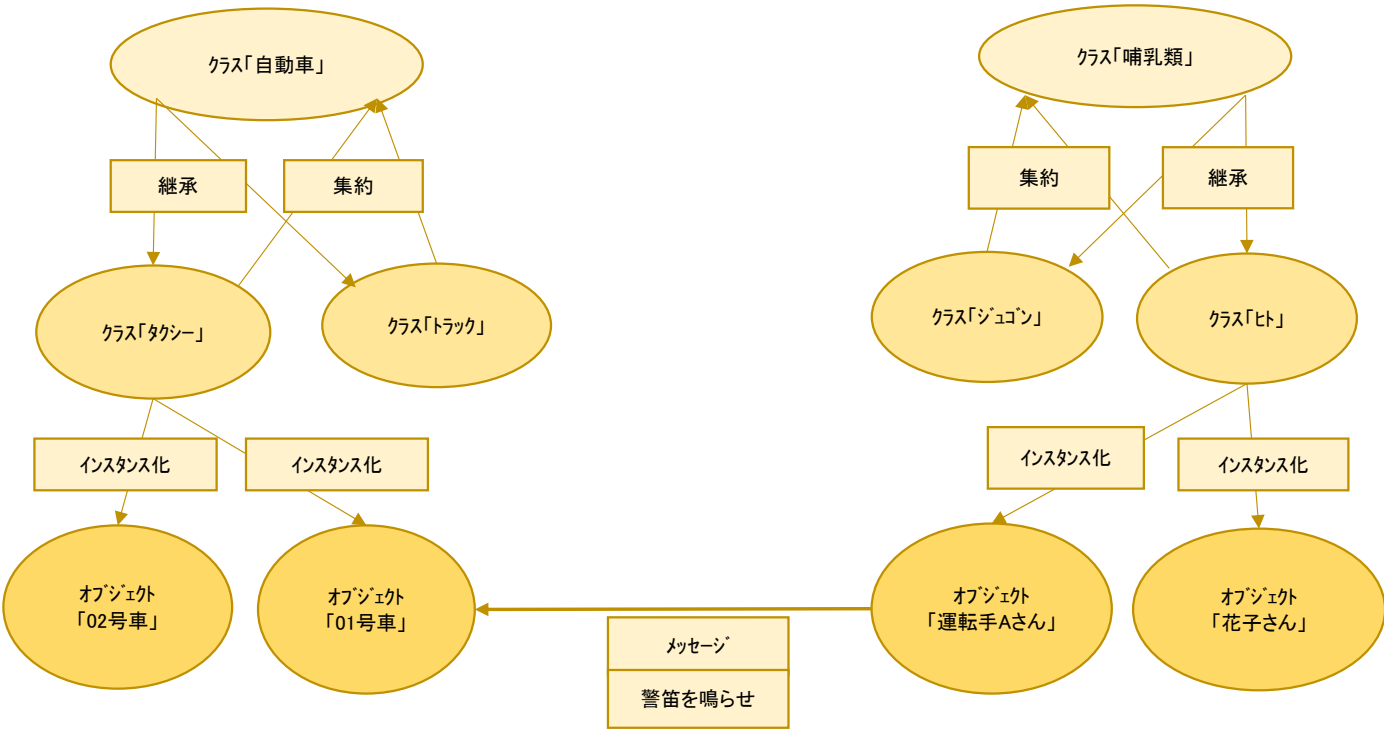
(14.1) オブジェクト指向とは

- ・オブジェクト指向（オブジェクトしこう “object-oriented”）とは、
オブジェクト同士の相互作用として、システムの振る舞いとらえる考え方です。
- ・オブジェクト指向プログラミングを構成する概念は次のようなものです。
 - (1) 「カプセル化」による情報隠蔽
 - (2) 「クラス」化による抽象化
 - (3) 「継承」による多態性（ポリモフィズム）
- ・以下に、オブジェクト指向で出てくる主な用語を掲載します。

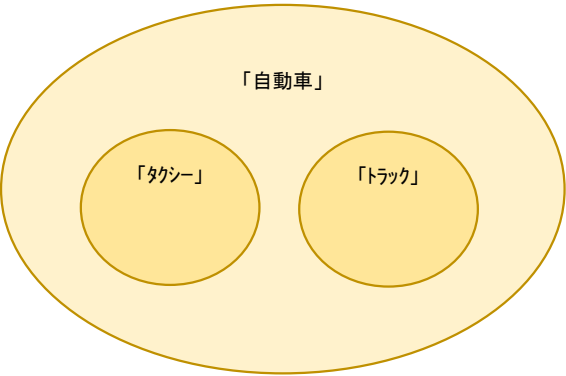
用語	意味
クラス	複数のデータを一つのまとまりとして「データ型」を定義します。 これは(int, char等の)「基本データ型」と区別して「抽象データ型」と呼ばれます。 「抽象データ型」は、「クラス」という用語で表現されます。
フィールド (属性)	「クラス」に含まれるそれぞれのデータは、「フィールド(属性)」と呼ばれます。 Python では「インスタンス」の属性としてアクセスする「インスタンス変数」と「インスタンス」に関係なく「クラス」共通の属性としてアクセスする「クラス変数」に分類されます。
メソッド	「クラス」には「フィールド」だけでなく、機能や処理手続きも含まれます。 「クラス」に付属する機能や処理手続きは、「メソッド」と言います。
メンバ	「クラス」の「フィールド」と「メソッド」を「クラス」の「メンバ」と言います。
オブジェクト	「クラス」を実体化(インスタンス化)したものを「オブジェクト」と言います。 Python における オブジェクト (object) とは、データを抽象的に表したものです。Python プログラムにおけるデータは全て、オブジェクトまたはオブジェクト間の関係として表されます。プログラムコード もまたオブジェクトとして表されます。
インスタンス	「クラス」を実体化(インスタンス化)したものを「インスタンス」とも言います。
メッセージ	オブジェクト指向において、オブジェクト間のコミュニケーションを「メッセージ」と呼びます。 「メッセージ」は、Pythonでは「メソッド」により実装されます。
カプセル化	オブジェクト指向において「データ型」の「属性」への直接的なアクセスを避け、 「メッセージ」を通して「属性」へのアクセスする方式を「カプセル化」と言います。 これは「情報隠蔽」に相当し、属性へのアクセスを保護します。
基本クラス 派生クラス	「クラス」間の共通項を抽出することにより、より上位の「クラス」を定義することが出来ます。この上位のクラスを「基本クラス (Base Class)」と言います。 「基本クラス (Base Class)」に対し、下位のクラスを「派生クラス (Derived Class)」と言います。 (「基本クラス」は「親クラス」、「スーパークラス」、「基底クラス」とも言います。 「派生クラス」は「子クラス」、「サブクラス」とも言います。)
継承	「派生クラス」は、その「基本クラス」の「フィールド」や「メソッド」を「継承」します。 「継承」は「is a」の関係性です(例:「ヒト」is a「哺乳類」)。 Java では単一のクラスのみを継承できるという「単一継承」の実装ですが、 Python では複数のクラスを継承できるという「多重継承」が実装できます。
集約	「基本クラス」は、その「派生クラス」の「フィールド」や「メソッド」を「集約」します。 「集約」は「has a」の関係性です(例:「哺乳類」has a「ヒト」)。
多態性	「多態性(ポリモフィズム)」とは、「基本クラス」から継承した「メソッド」を「オーバーライド」することで同じ「メソッド」を呼び出しても、実際の「オブジェクト」毎にその挙動を変えることが出来るというものです。
オーバーライド override	オーバーライドとは、「派生クラス」において「基本クラス」から継承された「メソッド」を独自の機能で上書きすることです。
オーバーロード over load	オーバーロードとは、同じクラスにおいて引数や戻り値が異なるが名称が同一の「メソッド」を定義することです。

(14. 2) オブジェクト指向での各要素の関連図

オブジェクト指向での各要素の関連の例を、以下に図示します：

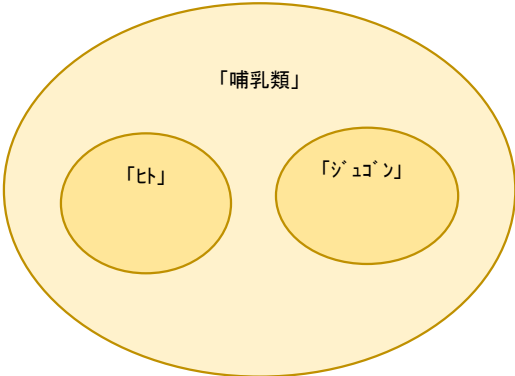


クラス間の継承関係は、集合の包含関係との対比で理解しやすくなります。
オブジェクトは集合の要素と見ることも出来ます。
例えば、上記の例は、以下のような集合の包含関係になります：



「自動車」 ⊃ 「タクシー」

「タクシー」 ⇒ 「01号車」
「タクシー」 ⇒ 「02号車」



「哺乳類」 ⊃ 「ヒト」

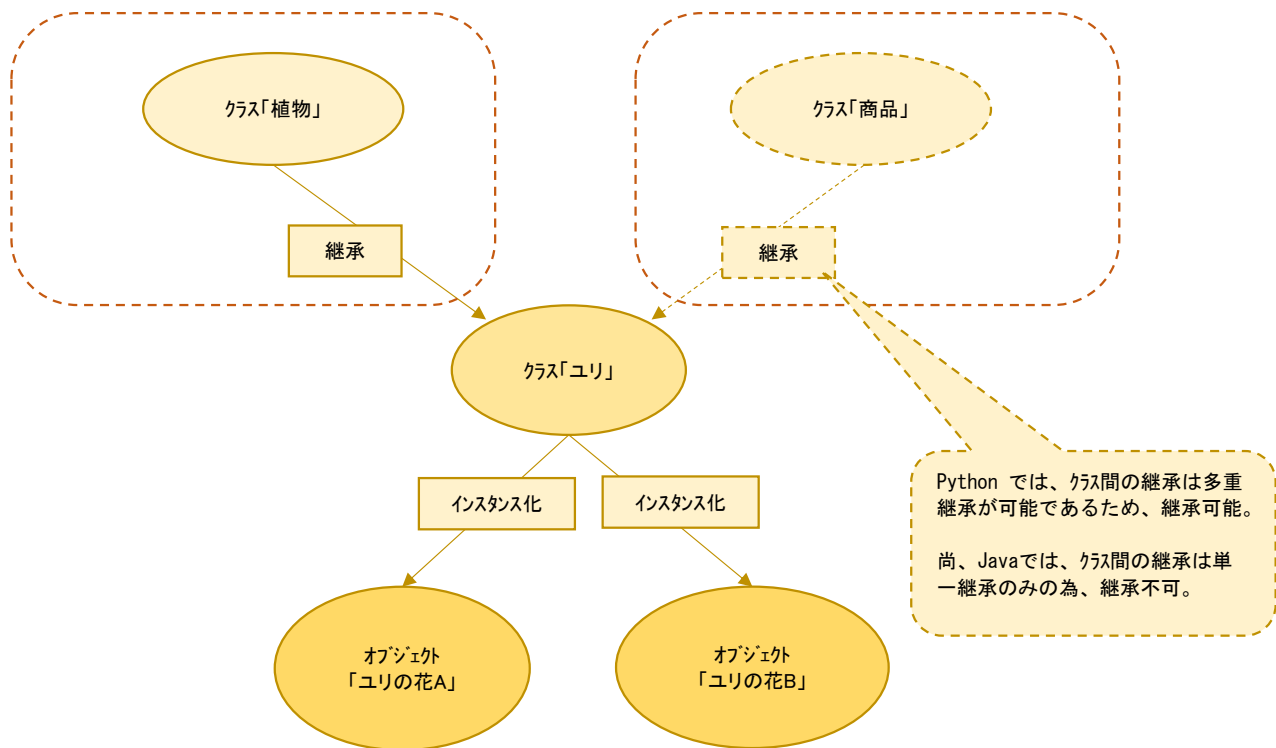
「ヒト」 ⇒ 「運転手Aさん」
「ヒト」 ⇒ 「花子さん」

【出典・参考】

⇒ <https://ja.wikipedia.org/wiki/オブジェクト指向>
「オブジェクト指向ソフトウェア工学」 海文堂 B・ヘンダーソン・セラーズ 著 大森健児訳 1994年
「パーフェクト C#」 技術評論社 醍醐竜一/斎藤友男著 2004年

(14.3) 単一継承と多重継承

- ・ クラス間で一つのスーパークラスから継承をすることを「単一継承」と言います。
 クラス間で複数のスーパークラスから継承をすることを「多重継承」と言います。
- ・ Pythonでは、クラス間での継承は「多重継承」が可能です。
- ・ 参考までに、Javaでは、クラス間での継承は「単一継承」しかできません。
 その代わりに、インターフェイスという実装で、多重継承に相当する仕組みを一部実現しています。



【出典・参考】

⇒ <http://www.ohshiro.tuis.ac.jp/~ohshiro/progaa/11/main00.html>

(14.4) Python プログラムのオブジェクト指向の実装

Python プログラムのオブジェクト指向の実装について、

⇒ https://docs.python.org/ja/3.6/reference/compound_stmts.html#class-definitions

⇒ http://www.k-techlabo.org/www_python/python_main.pdf

などを参照・引用しながら、その要約を記します。

(14.4.1) クラス定義

- ・クラス定義は、クラスオブジェクトを定義します。

(構文)

```
classdef ::= [decorators] "class" classname [inheritance] ":" suite
inheritance ::= "(" [argument_list] ")"
classname ::= identifier
```

(実装パターン)

```
class クラス名 ( 基底クラス1, 基底クラス2, . . . , 基底クラスn ) :
    ...文の記述...
```

- ・継承リスト(inheritance)には通常、基底クラスリストを指定します。
基底クラスリストは、一個以上の基底クラスをコンマ ',' で区切ったものです。
(Pythonでは多重継承が可能です。)
- ・継承リストを省略したクラスは、デフォルトで、基底クラス「object」を継承します。
- ・「...文の記述... (suite)」は、キーワード「class」よりも右の位置に同一の深さの字下げをして記述します。

(14.4.2) コンストラクタ

(14.4.2.1) コンストラクタの定義

- ・クラスのコンストラクタは、クラスのインスタンスを生成するものです。
- ・コンストラクタは、クラスの定義内に次のように「__init__」を実装します。

```
def __init__( self, 仮引数リスト ) :
    ...文の記述...
```

- ・「...文の記述...」は、キーワード「def」よりも右の位置に同一の深さの字下げをして記述します。
- ・「仮引数リスト」は、仮引数をカマ ',' 区切りで並べたもので省略可能です。
- ・キーワード「self」は生成するインスタンス自身を指しており、第1仮引数として記述します。
- ・基底クラスを持つクラスのコンストラクタ内では、...文の記述...として、以下の文を記述して基底クラスのコンストラクタを呼び出す形にします。

```
super().__init__(引数の列)
```

(14.4.2.2) クラスのインスタンスの生成

- ・クラスのインスタンスの生成は次のように行います。

```
インスタンス名 = クラス名 ( 実引数リスト )
```

- ・「実引数リスト」にはコンストラクタの「__init__」に記述した「self」より右の仮引数リストに与えるものを記述します。

(14.4.3) メソッド

- ・メソッドには、インスタンスメソッドとクラスメソッドがあります。

(14.4.3.1) インスタンスメソッド

- ・インスタンスメソッドは、クラスのインスタンス毎に呼び出すメソッドです。
- ・インスタンスメソッド定義は、コンストラクタの記述と似ています。

```
def メソッド名( self, 仮引数リスト ):
    ...文の記述...
```

- ・「…文の記述…」は、キーワード「def」よりも右の位置に同一の深さの字下げをして記述します。
- ・「仮引数リスト」は、仮引数をかマ' ,' 区切りで並べたもので省略可能です。
- ・キーワード「self」は生成するインスタンス自身を指しており、第1仮引数として記述します。

- ・インスタンスメソッドの参照は、ドット演算子を用いて行います。

```
戻り値リスト = インスタンス名. インスタンスメソッド名( self, 実引数リスト )
```

(14.4.3.2) クラスメソッド

- ・クラスメソッドは、クラスのインスタンスに関係なくクラス全体で共通のメソッドです。
- ・クラスメソッドは、クラスのメソッド定義の1行上にキーワード「def」と同じ字下げ位置にキーワード「@classmethod」という修飾子(decorator)を記述して定義します。

```
@classmethod
def メソッド名( self, 仮引数リスト ):
    ...文の記述...
```

- ・クラスメソッドの参照は、ドット演算子を用いて行います。

```
戻り値リスト = クラス名. クラスメソッド名( self, 実引数リスト )
```

(14.4.4) メンバ変数

- ・メンバ変数には、インスタンス変数とクラス変数があります。

(14.4.4.1) インスタンス変数

- ・インスタンス変数は、各インスタンス固有の変数です。
- ・Pythonではメンバ変数（フィールド）の存在を明に宣言するのではなく、そのクラスのコンストラクタやメソッドの定義の中で、「self」（つまり生成されたインスタンス自身）の属性として値を与えることでインスタンス変数を生成します。

```
def メソッド名( self, 仮引数リスト ):  
    self.インスタンス変数名1 = 値1  
    self.インスタンス変数名2 = 値2
```

- ・インスタンス変数の参照は、インスタンス名にドット演算子を付けて行います。

```
インスタンス名.インスタンス変数名
```

(14.4.4.2) クラス変数

- ・クラス変数は、クラスのインスタンスに関係なくクラス全体で共通の変数です。
- ・クラス変数の定義は、値の設定などオブジェクトの生成をクラスの定義の中に記述することで行います。

```
class クラス名 :  
    クラス変数名1 = 初期値1  
    クラス変数名2 = 初期値2
```

- ・クラス変数の参照は、クラス名にドット演算子を付けて行います。

```
クラス名.クラス変数名
```

(14.5) Python でのオブジェクト指向の実装例

- ・ 以下にオブジェクト指向での実装例を示します。

```
(実装例) #*****
# リスト03-(14)-1_オブジェクト指向での実装例
#*****
# クラス定義 -----
class AirPlane :
    # クラス変数
    clsName = "飛翔体のクラス"

    # コンストラクタ */
    def __init__( self, name, kind ):
        # インスタンス変数
        self.myName = name
        self.myWingKind = kind

    # インスタンスメソッド
    def Introduce(self, prefix, postfix):
        print(prefix, self.myName, postfix, self.myWingKind, '型の飛翔体', postfix)

    # クラスメソッド
    @classmethod
    def IntroduceClass(self):
        print(' このクラスは', AirPlane.clsName, 'です。')

# 呼出し側 -----

# クラスのインスタンスを作成
plane01 = AirPlane("スーパーマン", "宇宙人");
plane02 = AirPlane("ゼロ戦", "単葉機");
plane03 = AirPlane("スパイダーマン", "超人");

# インスタンスメソッド呼び出し
plane01.Introduce("吾輩は", "である。")
plane02.Introduce("私は", "です。")
plane03.Introduce("我は", "であるぞ。")

# クラス変数のアクセス
print(' クラス名=', AirPlane.clsName )

# クラスメソッド呼び出し
AirPlane.IntroduceClass()
```

```
(実行結果) 吾輩は スーパーマン である。 宇宙人 型の飛翔体 である。
私は ゼロ戦 です。 単葉機 型の飛翔体 です。
我は スパイダーマン であるぞ。 超人 型の飛翔体 であるぞ。
クラス名= 飛翔体のクラス
このクラスは 飛翔体のクラス です。
```

(15) 入出力

Python プログラムの入出力について、

サイト http://www.k-techlabo.org/www_python/python_main.pdf

「Pythonで動かして学ぶ！ あたらしい機械学習の教科書」（2018年01月 翔泳社 伊藤真著）

などを参照・引用しながら、その要約を記します。

(15.1) 標準出力／標準エラー出力

- ・ 通常の場合、標準出力／標準エラー出力はディスプレイを示しています。

(15.1.1) print関数

- ・ print関数は標準出力に対して出力するものであり、引数として与えた値を順番に標準出力に出力します。
print関数で出力すると行末で改行されます。
改行したくない場合はエンドオプションに"" (空文字) を指定「end=""」します。

```
print( 文字列オブジェクト [, エンドオプション] )
```

- ・ print関数で、format 関数を用いて文字列編集することも可能です。

(実装例)

```
x = 1 / 4
y = 2 / 4
z = 3 / 4
print('weight: {0:.3f} kg, {1:.3f} kg, '.format(x, y), end="")
print('{0:.3f} kg'.format(z))
```

(結果例)

```
weight: 0.250 kg, 0.500 kg, 0.750 kg
```

(15.1.2) sys.stdout オブジェクト

- ・ sys.stdout オブジェクトで、標準出力に出力することができます。
- ・ オブジェクトsys.stdoutは、標準出力を示すオブジェクトで、writeなどのメソッドで出力処理ができます。
出力処理が正常に終了すると、出力したバイト数が返されます。

```
import sys
戻り値 = sys.stdout.write( 文字列オブジェクト )
```

- ・ ioモジュールを使用することで、sys.stdout に writeメソッドで文字列を出力する際のエンコーディングを設定（「encoding=」に続けて文字コードを指定、例：'utf-8'）することができます。

```
sys.stdout = io.TextIOWrapper( sys.stdout.buffer, encoding='文字コード' )
```

(15.1.3) sys.stderr オブジェクト

- ・ sys.stderr オブジェクトで、標準エラー出力に出力することができます。
- ・ オブジェクトsys.stderrは、標準エラー出力を示すオブジェクトで、writeなどのメソッドで出力処理ができます。出力処理が正常に終了すると、出力したバイト数が返されます。

```
import sys
戻り値 = sys.stderr.write( 文字列オブジェクト )
```

- ・ ioモジュールを使用することで、sys.stderr に writeメソッドで文字列を出力する際のエンコーディングを設定（「encoding=」に続けて文字コードを指定、例：'utf-8'）することができます。

```
sys.stderr = io.TextIOWrapper( sys.stderr.buffer, encoding='文字コード' )
```

(15.1.4) 出力先のリダイレクト

- ・標準出力と標準エラー出力は同じ働きをしているように見えますが、これら2種類の出力は別のものであり、出力先のリダイレクトにより、別の出力先に送り出すことができます。
- ・リダイレクトとその出力先を一覧に示します。

リダイレクト	標準出力の出力先	標準エラー出力の出力先
(指定なし)	画面	画面
> 出力ファイル	出力ファイル	画面
1> 出力ファイル	出力ファイル	画面
2> 出力ファイル	画面	出力ファイル
>& 出力ファイル	出力ファイル	出力ファイル
&> 出力ファイル	出力ファイル	出力ファイル
> 出力ファイル 2>&1	出力ファイル	出力ファイル
1> 出力ファイル1 2> 出力ファイル2	出力ファイル1	出力ファイル2

(※ 出力を破棄する場合、出力ファイルに「/dev/null」を指定します)

(15.2) 標準入力

- ・ 通常の場合、標準入力はキーボードを示しています。

(15.2.1) input関数

- ・ input関数を呼び出すと、「プロンプト文字列」を標準出力に表示して入力を待ち、
1 行分の入力と改行入力により、1 行の内容を文字列として返します。

```
入力文字列オブジェクト = input( プロンプト文字列 )
```

(実装例)

```
#*****  
# リスト03-(15)-1_標準入力・標準出力  
#*****  
# 文字列として扱う  
xs = input(' 入力xs> ' )  
ys = input(' 入力ys> ' )  
print( xs+ys )  
  
# 整数として扱う  
xi = int(input(' 入力xi> '))  
yi = int(input(' 入力yi> '))  
print( xi+yi )
```

(結果例)

```
入力xs> 111  
入力ys> 222  
111222  
入力xi> 111  
入力yi> 222  
333
```

(15.2.2) sys.stdin オブジェクト

- ・ sys.stdinオブジェクトで、標準入力から入力することができます。
- ・ オブジェクトsys.stdinは標準入力を示すオブジェクトで、read、readlineなどのメソッドで入力処理ができます。入力処理が正常に終了すると、入力したデータを文字列型データとして返します。

```
import sys  
入力文字列 = sys.stdin.readline()
```

- ・ ioモジュールを使用することで、sys.stdin で文字列を入力する際のエンコーディングを設定（「encoding=」に続けて文字コードを指定、例：'utf-8'）することができます。

```
sys.stdin = io.TextIOWrapper( sys.stdin.buffer, encoding=' 文字コード' )
```

- ・ sys.stdin オブジェクトのメソッドを一覧に示します。

メソッド	機能
sys.stdin.readline	入力したデータ（1行）を受け取って表示します。
sys.stdin.readlines	入力したデータ（複数行）を受け取って表示します。 受け取ったデータは、リストに格納されます。
sys.stdin.read	入力したデータ（複数行）を受け取って表示します。 受け取ったデータは、1つの文字列に格納されます。

(15.3) ファイルへのアクセス

- ・ファイルへアクセスするには、ファイルオブジェクト(file object)を使用します。
- ・ファイルからの入力に先立って、open 関数を使用してファイルを開き、そのファイルに対応するファイルオブジェクト(file object)を生成しておきます。
以後はそのファイルオブジェクトで、ファイルへアクセスします。

(15.3.1) open関数

- ・ファイルからの入力に先立って、open 関数を使用してファイルを開きます。
ファイルのオープンが成功すると、そのファイルのファイルオブジェクトを返します。

```
ファイルオブジェクト = open( ファイルパス, モード [, encoding=文字コード] )
```

- ・「パス」は開く対象のファイルのパスを表す文字列型オブジェクト
- ・「モード」は開く対象のファイルへのアクセスモード
- ・「encoding=」に続けて文字コードを指定（例：'utf-8'）することができます。

- ・モードを一覧に示します。

モード	意味
r	読取り用（入力用）にファイルを開きます。
w	書き込み用（出力）にファイルを開きます。
a	追記用（出力用）にファイルを開きます。
r+	入出力両用にファイルを開きます。
b	ファイルは通常はテキスト形式として開かれますが、 上記モードに 'b' を書き加えると、バイナリ形式の扱いとなります。

- ・文字コードを一覧に示します。

文字コード	意味
utf-8	UTF-8
shift-jis	シフトJIS
cp932	シフトJIS（マイクロソフト標準キャラクタセットに基づく指定）
euc-jp	EUC
iso2022-jp	JIS

(15.3.2) close関数

- ・ファイルへのアクセスが終了したら、close 関数を使用して開かれているファイルを閉じます。

```
ファイルオブジェクト.close()
```

(15.3.3) テキストファイルからの読み込み

- ・テキストファイルから1行ずつデータを読み込むのが、readline 関数です。
- ・文字列オブジェクトの行末の改行コードと空白を削除するには、rstripメソッドを使用します。

```
1行分の文字列オブジェクト = ファイルオブジェクト.readline()
```

(実装例)

```
fObj = open(ファイルパス, 'r')
while True:
    line = fObj.readline().rstrip()
    if line:
        print(line)
    else:
        break
fObj.close()
```

(15.3.4) バイナリファイルからの読み込み

- ・バイナリファイルからデータを読み込むのが、read 関数です。
- ・サイズを引数に指定すると、指定したバイトサイズ分をファイルから読み込みます。
引数を指定しないと、ファイル全体をファイルから読み込みます。

```
読みこんだファイル全体分の文字列オブジェクト = ファイルオブジェクト.read()  
または  
読み込んだバイト列 = ファイルオブジェクト.read(読み込むバイトサイズ)
```

(実装例)

```
fobj = open("E:\¥99_temp¥¥aaa.txt", 'rb')  
while True:  
    block = fobj.read(100)  
    if block:  
        print(block)  
    else:  
        break  
fobj.close()
```

(15.4) その他のファイルアクセス

(15.4.1) 「numpy (ナンパイ)」を用いたファイルへの保存・読み込み

- ・「numpy (ナンパイ)」というライブラリで、ベクトルや行列のファイルへの保存・読み込みができます。

- ・1つのndarray 型をファイルに保存するには、save 関数を使います（ファイルの拡張子は、.npy）。

```
np.save('ファイル名.npy', 変数名)
```

その読み出しは、load 関数を使います。

```
np.load('ファイル名.npy')
```

(実装例)

```
#####  
# リスト03-(15)-2_numpyを用いたファイルへの保存・読み込みその1  
#####  
import numpy as np  
  
data = np.array([1, 2, 3, 4, 5, 6, 7])  
print('保存前:', data)  
np.save('datafile.npy', data) # 保存  
data = [] # データの消去  
print('消去後:', data)  
data = np.load('datafile.npy') # 読み込み  
print('読み込み後:', data)
```

(結果例)

```
保存前: [1 2 3 4 5 6 7]  
消去後: []  
読み込み後: [1 2 3 4 5 6 7]
```

- ・複数のndarray 型を保存するには、savez 関数を使います（ファイルの拡張子は、.npz）。

```
np.savez('ファイル名.npz', 変数名1=変数名1, 変数名2=変数名2, ...)
```

- ・そのデータを load 関数でロードすると、
保存したすべての変数がロード先の変数（下例では outfile）に格納され、
ロード先の変数['変数名']によって、それぞれの変数を参照することができます。
ロード先の変数.files で格納してある変数のリストを見ることができます。

(実装例)

```
#####  
# リスト03-(15)-3_numpyを用いたファイルへの保存・読み込みその2  
#####  
import numpy as np  
  
data1 = np.array([11, 12, 13])  
data2 = np.array([21, 22, 23])  
  
# データの保存  
np.savez('datafile2.npz', data1=data1, data2=data2)  
# データの消去  
data1 = []  
data2 = []  
print('消去後 data1:', data1)  
print('消去後 data2:', data2)  
  
# データの読み込み  
outfile = np.load('datafile2.npz')  
print('読み込み後 files:', outfile.files) # 格納データの表示  
data1 = outfile['data1'] # data1 の取り出し  
data2 = outfile['data2'] # data2 の取り出し  
print('読み込み後 data1:', data1)  
print('読み込み後 data2:', data2)
```

(結果例)

```
消去後 data1: []  
消去後 data2: []  
読み込み後 files: ['data1', 'data2']  
読み込み後 data1: [11 12 13]  
読み込み後 data2: [21 22 23]
```

(16) インタープリタ

Python インタープリタについて、下記サイト

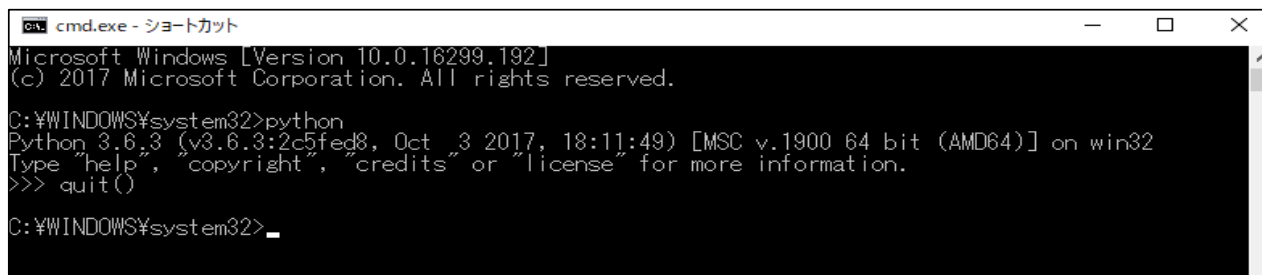
⇒ <https://docs.python.org/ja/3.6/using/cmdline.html>

⇒ http://www.k-techlabo.org/www_python/python_main.pdf

などを参照・引用しながら、その要約を記します。

(16.1) インタープリタの起動・終了

- ・ Python インタープリタ はDOS窓で、「python」で起動し、「quit()」で終了します。

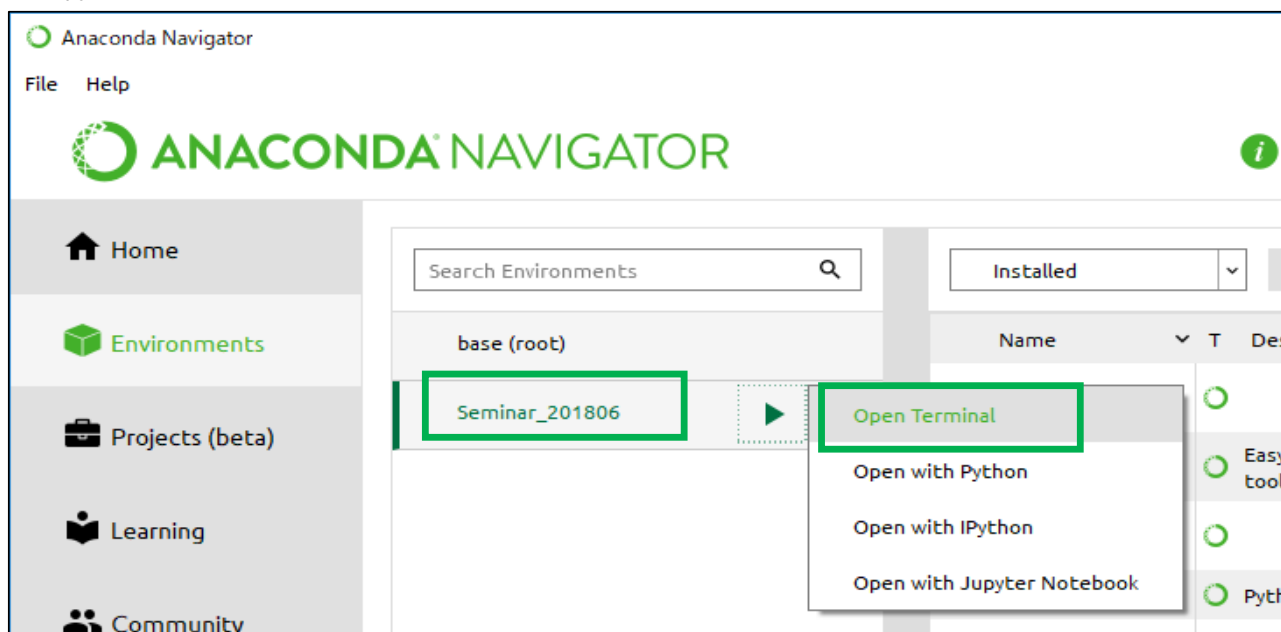


```
cmd.exe - ショートカット
Microsoft Windows [Version 10.0.16299.192]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>python
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 18:11:49) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> quit()

C:\WINDOWS\system32>
```

- ・ Anaconda Navigator からの場合は、「環境名 → Open Terminal」でDOS窓を開いた上で、「python」を起動します。



(16.2) コマンド仕様

- Python インタプリタのコマンド仕様の詳細を以下に記します：

(構文)

```
python [-bBdEhiIOqsSuvVWx?] [-c command | -m module-name | script | - ] [args]
```

- Pythonインタプリタコマンドの主なオプションを以下に記します：

No.	書式	意味
1	-c command	<ul style="list-style-type: none">コマンド (command) 内の Python コードを実行します。command は改行によって区切られた1行以上の文です。通常のモジュールのコードと同じく、行頭の空白文字は意味を持ちます。
2	-m module-name	<ul style="list-style-type: none">sys.path から指定されたモジュール名 (module-name) のモジュールを探し、その内容を __main__ モジュールとして実行します。引数はモジュール名なので、拡張子 (.py) を含めてはいけません。通常のモジュールの代わりにパッケージ名が与えられた場合、インタプリタは <pkg>.__main__ を main モジュールとして実行します。
3	script	<ul style="list-style-type: none">Python スクリプトファイルのパスを指定します。
4	?, -h, --help	<ul style="list-style-type: none">全コマンドラインオプションの短い説明を出力します。

(16.3) コマンドラインの引数渡し

- ソースプログラムをPython処理系（インタプリタ）にスクリプトとして与えて実行を開始する場合、起動時に与えたコマンド引数を取得するには sys モジュールのプロパティ argv を参照します。

(コマンド引数取得の実装パターン)

```
import sys

# コマンド引数の取得
print( "コマンド 引数=", sys.argv)
```

(17) 確認問題

以下の空欄に最もあてはまる用語を選択肢から選び、その記号を回答欄に記入してください。

(1) 【Python の言語としての特徴】

- Python は(1.1) _____ の(1.2) _____ で、初心者にお勧め出来る言語として、米国の大学で言語学習の導入段階において最も学ばれています。
- C 言語やJava は変数の使用に先立って変数の確保とデータ型を宣言する必要があります。これを(1.3) _____ と言います。これに対し、Python では、変数の使用に先立って変数の確保とデータ型を宣言する必要がありません。これを(1.4) _____ と言います。

- (選択肢) (a) 「動的な型付けの言語処理系」
(b) 「強い型付けの言語処理系」
(c) 手続き指向
(d) オブジェクト指向
(e) スクリプト言語
(f) コンパイラ言語

(回答)	(1.1)	
	(1.2)	
	(1.3)	
	(1.4)	

(2) 【Python の行構造】

- Pythonスクリプトの文字コードは、デフォルトでは(2.1) _____ になっています。デフォルト以外のエンコーディングを宣言するには、ファイルの最初の行（シェバンが第1行の場合は、第2行）としてエンコーディング宣言用の特別なコメント行を追加する必要があります。
- Pythonのコメントは、文字列リテラル内に入っていない(2.2) _____ で始まり、同じ物理行の末端で終わります。
- Pythonでは、二つ以上の物理行を論理行としてつなげるためには、(2.3) _____ を使います。丸括弧 (parentheses)、角括弧 (square bracket)、および波括弧 (curly brace) 内の式は、(2.3) _____ を使わずに一行以上の物理行に分割することができます。
- Python では、(2.4) _____ によって文などのかたまりの範囲（ブロック、for 文や関数定義など）を示します。これを(2.5) _____ と言います。同一ブロック内の各行は、同じ桁数分下げする必要があります。

- (選択肢) (a) ハッシュ文字 (#)
(b) バックスラッシュ文字 (\)
(c) 「オフサイドルール (Off-side Rule)」
(d) 「UTF-8」
(e) 「UNICODE」
(f) 「Shift-Jis」
(g) 対括弧 {}
(h) 字下げ

(回答)	(2.1)	
	(2.2)	
	(2.3)	
	(2.4)	
	(2.5)	

(3) 【リテラル】

- ・ 文字列リテラルとバイト列リテラルは、対応する (3.1) _____ または (3.2) _____ で囲みます。
- ・ また、対応する三連の (3.1) _____ や (3.2) _____ で囲うこともできます (これは、(3.3) _____ と呼ばれます)。
これにより、複数の行に渡る文字列を記述することができます。
- ・ 数値型オブジェクトは、数値リテラルによって作成されたり、算術演算や組み込みの算術関数によって返される数値を表すオブジェクトです。数値リテラルには、整数 (integer)、浮動小数点数 (floating point number)、虚数 (imaginary numbers) の 3 種類あります。

- (選択肢) (a) ハッシュ文字 (#)
(b) バックスラッシュ文字 (\)
(c) 一重引用符 (')
(d) 二重引用符 (")
(e) 「三重クオート文字列 (triple-quoted string)」

(回答)

(3.1)	
(3.2)	
(3.3)	

(4) 【オブジェクト型、特にシーケンス型 (Sequences)】

- ・ オブジェクト型の中で、シーケンス型 (Sequences) は、有限の順序集合 (ordered set) を表現します。
- ・ シーケンス型には、以下の2種類あります。
 - (1) 一度生成されるとその値を変更することができない「変更不能なシーケンス (immutable sequence)」
 - (2) 作成後に変更することができる「変更可能なシーケンス型 (mutable sequence)」

- ・ 変更不能なシーケンス型には以下の3種類あります。

(1) (4.1) _____

- ・ (4.1) _____ は Unicode コードポイントを表現する値の配列です。
- ・ Python には 文字型 が無い代わりに、
どの文字も長さ1の (4.1) _____ で文字列を表現します。

(2) (4.2) _____

- ・ (4.2) _____ の要素は任意の Python オブジェクトです。
- ・ 個々の要素を表現する式をカンマで区切り、全体を丸括弧 () で囲んで構成します。

(3) (4.3) _____

- ・ 要素はバイトで、 $0 \leq x < 256$ の範囲の整数で表現される不変な配列です。
- ・ バイト列リテラルや組み込みの `bytes()` コンストラクタを使って本オブジェクトを作成します。

- ・ 変更可能なシーケンス型には以下の2種類あります。

(1) (4.4) _____

- ・ (4.4) _____ の要素は任意の Python オブジェクトです。
- ・ 個々の要素を表現する式をカンマで区切り、全体を角括弧 [] で囲んで構成します。

(2) (4.5) _____

- ・ 要素はバイトで、 $0 \leq x < 256$ の範囲の整数で表現される変更可能な配列です。
- ・ 組み込みの `bytearray()` コンストラクタによって作成されます。

- (選択肢) (a) bytes
(b) タプル型 (tuple)
(c) バイト配列 (bytearray)
(d) リスト型 (list)
(e) 文字列型 (string)

(回答)

(4.1)	
(4.2)	
(4.3)	
(4.4)	
(4.5)	

(5) 【変数のスコープ】

- 関数の内部で生成したオブジェクトは、基本的にはその関数の(5.1)_____であり、その関数の実行が終了した後は消滅します。
- 関数の外部で生成された(5.2)_____を関数内部で更新するには、当該関数内で(5.2)_____の使用を宣言します。具体的には関数定義の内部で、「(5.3)_____ (5.2)_____の名前」と記述します。

- (選択肢) (a) 「ローカル変数 (局所変数)」
(b) 「大域変数」
(c) external
(d) global

(回答)

(5.1)	
(5.2)	
(5.3)	

(6) 【関数】

- 関数定義では、return 文で戻り値を返しますが、return 文を書かない場合、(6.1)_____ が返ります。
- return 文の戻り値としてカンマ「,」で区切ることにより、戻り値を複数指定できます。
その場合、呼び出し側は、受け取る変数をカンマ「,」で区切って複数並べることにより、戻り値を複数取得できます。あるいは、tuple 型として一つの変数で受け取ることも出来ます。
- (6.2)_____ 式は、匿名関数の作成に使用されます。

- (選択肢) (a) Null
(b) None
(c) 「アノニマス (anonymous)」
(d) 「ラムダ (lambda)」

(回答)

(6.1)	
(6.2)	

(7) 【モジュールとパッケージ】

- Python では定義をファイルに書いておき、他のスクリプトやインタプリタの対話インスタンス上で使う方法があります。このファイルを(7.1)_____と呼びます。
- (7.1)_____にある定義は、他の(7.1)_____に取り込むことができます。
その方法の一つが、(7.2)_____によるものです。
- (7.1)_____の組織化を助け、名前階層を提供するために、Python には(7.3)_____という手法があります。物理的にはフォルダ構成として、階層的に(7.1)_____のファイル群を配置します。

- (選択肢) (a) 「コンポーネント (component)」
(b) 「パッケージ (package)」
(c) 「モジュール (module)」
(d) export文
(e) import文

(回答)

(7.1)	
(7.2)	
(7.3)	

(8) 【オブジェクト指向】

- クラス間で一つのスーパークラスから継承をすることを(8.1)_____と言います。
クラス間で複数のスーパークラスから継承をすることを(8.2)_____と言います。
- クラス間での継承について、(8.2)_____の指定がPythonではできますが、Javaではできません。
その代わりに、Javaではインターフェイスという実装で、(8.2)_____に相当する仕組みを一部実現しています。

- (選択肢) (a) 「多重継承」
(b) 「単一継承」

(回答)

(8.1)	
(8.2)	

(18) 確認問題解答用紙

提出者 :
提出日 : 年 月 日

回答

No.	回答
(1. 1)	
(1. 2)	
(1. 3)	
(1. 4)	
(2. 1)	
(2. 2)	
(2. 3)	
(2. 4)	
(2. 5)	
(3. 1)	(※1)
(3. 2)	(※1)
(3. 3)	
(4. 1)	
(4. 2)	
(4. 3)	
(4. 4)	
(4. 5)	

(※1) 3. 1, 3. 2は順不問

No.	回答
(5. 1)	
(5. 2)	
(5. 3)	
(6. 1)	
(6. 2)	
(7. 1)	
(7. 2)	
(7. 3)	
(8. 1)	
(8. 2)	

採点結果

 / 27

(※黄色の枠のみ記入をお願いします)

※ ご意見・ご要望などありましたら、下欄に記してください。

(資料)

(資料1) コーディング・スタイル

Pythonコードのスタイルガイドが

⇒ <https://pep8-ja.readthedocs.io/ja/latest/>

⇒ <https://github.com/mumumu/pep8-ja>

⇒ <https://www.python.org/dev/peps/pep-0008/>

にあります。実装の際には、このガイドに準拠した方がよいでしょう。

以下に幾つか抜粋を記します：

- ・ 4桁のインデントを使用し、タブは使用しないでください。
- ・ 79文字を超えないように行を折り返します。
- ・ トップレベルの関数やクラスは、2行ずつ空けて定義するようにしてください。
- ・ 関数とクラスを分離するには空白行を使用し、関数内には大きなコードブロックを使用します。
- ・ import文は、通常は行を分けるべきです：
- ・ 可能であれば、独自の行にコメントを付けてください。
- ・ docstringsを使用してください。
- ・ 演算子の前後、カンマの後にはスペースを入れてください。
- ・ Pythonのデフォルト、UTF-8、さらには平易なASCIIは、どんな場合でも最適です。
- ・ 異なる言語を話す人々がコードを読んだり保守したりする可能性がある場合には、識別子にASCII以外の文字を使用しないでください。

・ 命名規約は以下のとおり：

タイプ	命名規約
定数	英大文字・数とアンダースコア
変数	英小文字・数とアンダースコア “_”。最初の文字は数字以外。
プライベート変数	最初に_をつけて、英小文字・数とアンダースコア
関数	英小文字・数とアンダースコア
引数	英小文字・数か、英小文字・数とアンダースコア
プロパティ	英小文字・数とアンダースコア
クラス	Camel形式（複合語の語頭を大文字にする記法）
モジュールとパッケージ	英小文字・数

※ 小文字のエル “l”、大文字のオー “O”、大文字のアイ “I” は、一文字変数名として使用しないでください。

(資料2) リストオブジェクトのメソッド一覧

- ・ リストオブジェクトのメソッドについて、
⇒ <https://docs.python.org/ja/3/tutorial/datastructures.html>
などを参照・引用しながら、その一覧を記します。

(※) リストオブジェクト名を “list” として説明しています。

メソッド	機能
<code>list.append(x)</code>	リストの最後に項目を追加します。 <code>list[len (list) :] = [x]</code> と等価です。
<code>list.extend(iterable)</code>	イテラブルからすべての項目を追加してリストを拡張します。 <code>list[len (list) :] = iterable</code> に相当します。
<code>list.insert(i, x)</code>	指定された位置に項目を挿入します。最初の引数は、挿入する前の要素のインデックスです。したがって、 <code>a.insert (0, x)</code> はリストの先頭に挿入されます。 <code>list.insert (len (list) , x)</code> は、 <code>list.append(x)</code> と等価です。
<code>list.remove(x)</code>	値がxである最初の項目をリストから削除します。そのような項目がない場合はエラーです。
<code>list.pop([i])</code>	リスト内の指定された位置にある項目を削除して返します。インデックスが指定されていない場合、 <code>list.pop ()</code> はリスト内の最後の項目を削除して返します。 (メソッドのシグネチャのiを囲む角かっこは、そのパラメータがオプションであることを示しています。その位置に角括弧を入力する必要はありません。この表記は、Pythonライブラリリファレンスによく表示されます)。
<code>list.clear()</code>	リストからすべての項目を削除します。 <code>del list [:]</code> と同じです。
<code>list.index(x[, start[, end]])</code>	値がxである最初の項目のリスト内のゼロから始まるインデックスを返します。そのような項目がない場合、 <code>ValueError</code> を発生させます。 オプションの引数startとendは、スライス表記法のように解釈され、検索をリストの特定のサブシーケンスに限定するために使用されます。 返されるインデックスは、start引数ではなく、フルシーケンスの開始位置を基準として計算されます。
<code>list.count(x)</code>	x がリスト list に現れる回数を返します。
<code>list.sort(key=None, reverse=False)</code>	リストの項目をソートします (ソートのカスタマイズに使用できます。ソートの説明は <code>sorted()</code> を参照してください) 。
<code>list.reverse()</code>	リストの要素をその場で逆順にします。
<code>list.copy()</code>	リストのシャローコピーを返します。 <code>list[:]</code> と同じです。

(資料3) グラフ描画での色一覧

・matplotlib によるグラフ描画での、色一覧は以下のとおり：

No.	色名	RGB
1	aliceblue	#F0F8FF
2	antiquewhite	#FAEBD7
3	aqua	#00FFFF
4	aquamarine	#7FFFD4
5	azure	#F0FFFF
6	beige	#F5F5DC
7	bisque	#FFE4C4
8	black	#000000
9	blanchedalmond	#FFEBCD
10	blue	#0000FF
11	blueviolet	#8A2BE2
12	brown	#A52A2A
13	burlywood	#DEB887
14	cadetblue	#5F9EA0
15	chartreuse	#7FFF00
16	chocolate	#D2691E
17	coral	#FF7F50
18	cornflowerblue	#6495ED
19	cornsilk	#FFF8DC
20	crimson	#DC143C
21	cyan	#00FFFF
22	darkblue	#00008B
23	darkcyan	#008B8B
24	darkgoldenrod	#B8860B
25	darkgray	#A9A9A9
26	darkgreen	#006400
27	darkgrey	#A9A9A9
28	darkkhaki	#BDB76B
29	darkmagenta	#8B008B
30	darkolivegreen	#556B2F
31	darkorange	#FF8C00
32	darkorchid	#9932CC
33	darkred	#8B0000
34	darksalmon	#E9967A
35	darkseagreen	#8FBC8F
36	darkslateblue	#483D8B
37	darkslategray	#2F4F4F
38	darkslategrey	#2F4F4F
39	darkturquoise	#00CED1
40	darkviolet	#9400D3
41	deeppink	#FF1493
42	deepskyblue	#00BFFF
43	dimgray	#696969
44	dimgrey	#696969
45	dodgerblue	#1E90FF
46	firebrick	#B22222
47	floralwhite	#FFFAF0

No.	色名	RGB
48	forestgreen	#228B22
49	fuchsia	#FF00FF
50	gainsboro	#DCDCDC
51	ghostwhite	#F8F8FF
52	gold	#FFD700
53	goldenrod	#DAA520
54	gray	#808080
55	green	#008000
56	greenyellow	#ADFF2F
57	grey	#808080
58	honeydew	#F0FFF0
59	hotpink	#FF69B4
60	indianred	#CD5C5C
61	indigo	#4B0082
62	ivory	#FFFFFF
63	khaki	#F0E68C
64	lavender	#E6E6FA
65	lavenderblush	#FFF0F5
66	lawngreen	#7CFC00
67	lemonchiffon	#FFFACD
68	lightblue	#ADD8E6
69	lightcoral	#F08080
70	lightcyan	#E0FFFF
71	lightgoldenrodyellow	#FAFAD2
72	lightgray	#D3D3D3
73	lightgreen	#90EE90
74	lightgrey	#D3D3D3
75	lightpink	#FFB6C1
76	lightsalmon	#FFA07A
77	lightseagreen	#20B2AA
78	lightskyblue	#87CEFA
79	lightslategray	#778899
80	lightslategrey	#778899
81	lightsteelblue	#B0C4DE
82	lightyellow	#FFFFE0
83	lime	#00FF00
84	limegreen	#32CD32
85	linen	#FAF0E6
86	magenta	#FF00FF
87	maroon	#800000
88	mediumaquamarine	#66CDAA
89	mediumblue	#0000CD
90	mediumorchid	#BA55D3
91	mediumpurple	#9370DB
92	mediumseagreen	#3CB371
93	mediumslateblue	#7B68EE
94	mediumspringgreen	#00FA9A

No.	色名	RGB
95	mediumturquoise	#48D1CC
96	mediumvioletred	#C71585
97	midnightblue	#191970
98	mintcream	#F5FFFA
99	mistyrose	#FFE4E1
100	moccasin	#FFE4B5
101	navajowhite	#FFDEAD
102	navy	#000080
103	oldlace	#FDF5E6
104	olive	#808000
105	olivedrab	#6B8E23
106	orange	#FFA500
107	orangered	#FF4500
108	orchid	#DA70D6
109	palegoldenrod	#EEE8AA
110	palegreen	#98FB98
111	paleturquoise	#AFEEEE
112	palevioletred	#DB7093
113	papayawhip	#FFEFD5
114	peachpuff	#FFDAB9
115	peru	#CD853F
116	pink	#FFC0CB
117	plum	#DDA0DD
118	powderblue	#B0E0E6
119	purple	#800080
120	rebeccapurple	#663399
121	red	#FF0000
122	rosybrown	#BC8F8F
123	royalblue	#4169E1
124	saddlebrown	#8B4513
125	salmon	#FA8072
126	sandybrown	#F4A460
127	seagreen	#2E8B57
128	seashell	#FFF5EE
129	sienna	#A0522D
130	silver	#C0C0C0
131	skyblue	#87CEEB
132	slateblue	#6A5ACD
133	slategray	#708090
134	slategrey	#708090
135	snow	#FFFAFA
136	springgreen	#00FF7F
137	steelblue	#4682B4
138	tan	#D2B48C
139	teal	#008080
140	thistle	#D8BFD8
141	tomato	#FF6347
142	turquoise	#40E0D0
143	violet	#EE82EE

No.	色名	RGB
144	wheat	#F5DEB3
145	white	#FFFFFF
146	whitesmoke	#F5F5F5
147	yellow	#FFFF00
148	yellowgreen	#9ACD32

※ この表は以下の実行結果をもとにして作成した：

```
import matplotlib
matplotlib.colors.cnames
```