**[@kita\_ds12(Ryota KITA)](https://qiita.com/kita_ds12)**

<https://qiita.com/kita_ds12/items/84552d41a8aad36d5519>

**Python基礎文法まとめ**

* [Python](https://qiita.com/tags/python)
* [プログラミング](https://qiita.com/tags/%e3%83%97%e3%83%ad%e3%82%b0%e3%83%a9%e3%83%9f%e3%83%b3%e3%82%b0)
* [初心者](https://qiita.com/tags/%e5%88%9d%e5%bf%83%e8%80%85)
* [Python3](https://qiita.com/tags/python3)

最終更新日 2023年05月22日投稿日 2022年05月25日

**はじめに**

本記事は学内の勉強会用に作成した資料をそのまま貼り付けたものとなります。  
口頭での説明を前提として作成しているため、  
一部内容が不十分であったり、正確ではない表現が含まれているものがあります。

また、モジュールや例外処理など今後追加予定のものも多くあり、  
現時点で内容を網羅しているわけではないことをご了承ください。

**目次**

1. 行構造
   1. 式
   2. 文
   3. 節
   4. コメント
2. 入出力
   1. print関数
   2. エスケープシーケンス
   3. input関数
3. 算術演算
   1. リテラル
   2. 算術演算子
   3. 文字列演算
4. 変数
   1. 変数
   2. 命名規則
   3. 予約語
   4. 代入演算子
   5. 変数の演算
   6. 複合演算子
5. オブジェクト
   1. オブジェクト
   2. ID
   3. 型
6. 数値型
   1. int
   2. float
   3. complex
7. シーケンス
   1. シーケンス
   2. list
   3. tuple
   4. rangeオブジェクト
   5. string
8. マッピング
   1. dictionary
9. 集合
   1. set
   2. frozenset
10. 条件分岐
    1. 真偽値
    2. 比較演算子
    3. 所属検査演算子
    4. 論理演算子
    5. if文
    6. elif節
    7. else節
    8. 三項演算子
11. 繰り返し
    1. while文
    2. for文
12. 内包表記
    1. リスト内包表記
    2. セット内包表記
    3. 辞書内包表記
    4. ジェネレータ式
13. 関数
    1. 関数
    2. def文
    3. 引数
    4. return文
    5. 名前空間
    6. 関数内関数
    7. クロージャ
    8. lambda式
    9. 高階関数
14. 内包表記
    1. リスト内包表記
    2. セット内包表記
    3. 辞書内包表記
    4. ジェネレータ式
15. イテレータ
    1. イテレータ
    2. ジェネレータ
16. クラス
    1. オブジェクト指向
    2. class文
    3. クラス変数
    4. アトリビュート
    5. コンストラクタ
    6. メソッド
    7. デストラクタ
    8. 継承
    9. 多態性

**行構造**

**< 式 >**

**式** ... 文(命令)の構成要素であり, **値を返す** もの.

文字列: "apple"  
数値: 10  
演算式: 10+5  
変数: hoge  
リスト: [10, 20, 30]  
スライス: [10, 20, 30][1]  
呼び出し: abs(-10)  
※abs関数は絶対値を返す組み込み関数

**< 文 >**

**文** ... 命令そのものであり, **値は返さない**.  
Pythonにおける文は **単純文** と **複合文** に分かれている.  
**単純文** ... 単一の文実行される実行される命令.

代入文: hoge = 10  
pass文: pass  
del文: del hoge  
return文: return hoge  
break文: break  
continue文: continue  
import文: import math

**複合文** ... 包含する複数の文の制御に影響を与える命令.

if文

if hoge > 5:

pass

while文

while True:

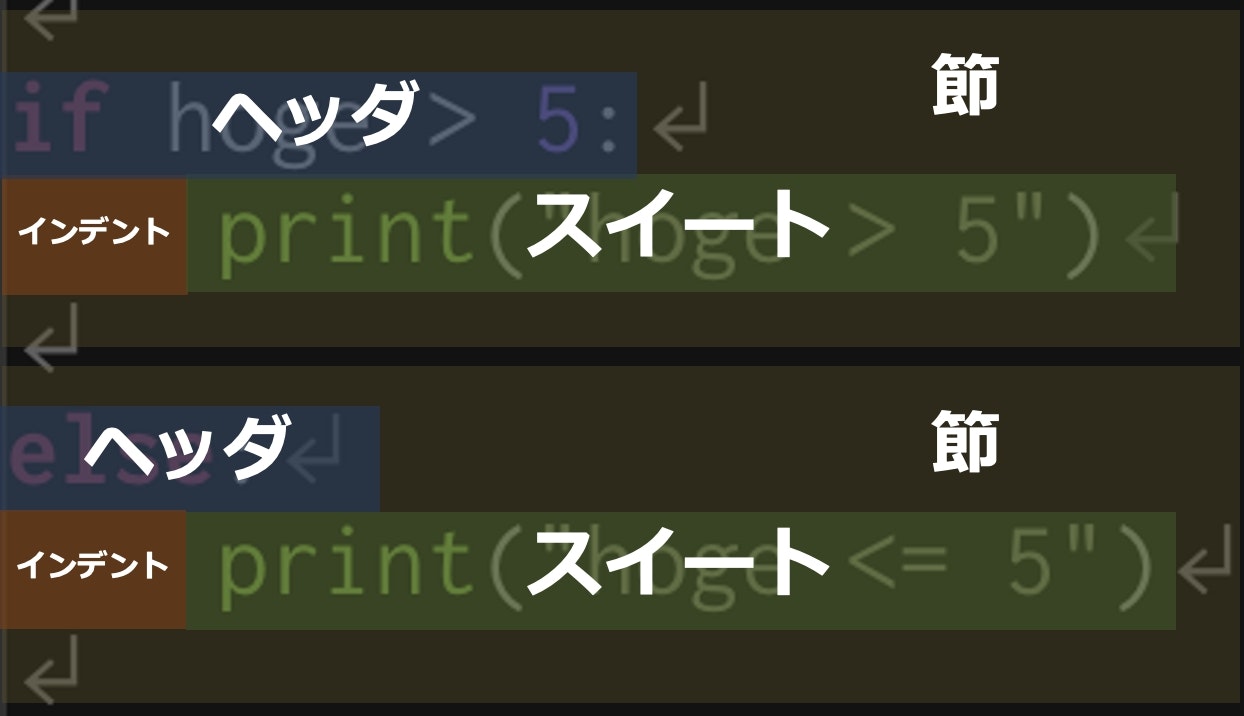
hoge += 1

for文

for fuga in hoge:

print(fuga)

**< 節 >**

**節** ... 複合文の実行単位であり, **ヘッダ** と **スイート** から構成される.  
**ヘッダ** ... 節の先頭に存在し, 処理を識別するキーワードから行末の:までを指す.  
**スイート** ... ヘッダ以降の文を指し, **インデント**(原則半角スペース4つ) で範囲を示す.  
[](https://camo.qiitausercontent.com/8258d83a25a925a3c333ab651feb0647906fc30d/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f33326431623564612d623034662d373565632d343461322d3533623933313830663932622e6a706567)

**< コメント >**

**コメント** ... # から行末までは実行時に無視される.  
主に保守などを目的としたプログラムに関するメモとして使用される.

行頭に# がつくコメントを **ブロックコメント** と呼ぶ.

# 実行可能な上限

hoge = 500

行の途中から始まるコメントを **インラインコメント** と呼ぶ.

hoge = 500 # 実行可能な上限

**入出力**

**< print関数 >**

**print関数** ...()内に指定した式を表示する.

print(5)

>> 5

print関数を並べると表示結果も縦に並ぶ.

print(1)

print(2)

print(3)

>> 1

2

3

オプション引数 end='' を設定することで改行させないようにできる.

print(1, end='')

print(2, end='')

print(3)

>> 123

本来は end='\n' が省略されている.

\n は **改行** を表す **特殊記号(エスケープシーケンス)** の1つ.

print関数は , で区切ることで複数の式を渡すことができる.  
渡した式はすべて **半角スペース** で区切られ, 横に並ぶ.

print(1,2,3)

>> 1 2 3

オプション引数 sep='' を設定することで区切り文字を変更できる.

print(1,2,3, sep='-')

>> 1-2-3

本来は sep=' ' (半角スペース)が省略されている.

**< エスケープシーケンス >**

**エスケープシーケンス** ... 文字を出力する際に, 改行などの **特殊な制御** を行う記号.

| **エスケープシーケンス** | **制御内容** |
| --- | --- |
| \n | 改行 |
| \t | 水平タブ |
| \v | 垂直タブ |
| \\ | バックスラッシュの表示 |
| \' | シングルクォーテーションの表示 |
| \" | ダブルクォーテーションの表示 |

\n ... 改行

print("1行目\n2行目")

>> 1行目

2行目

\t ... 水平タブ

print("1/1\t正月")

print("12/25\tクリスマス")

>> 1/1 正月

12/25 クリスマス

\v ... 垂直タブ

print("hoge\vhoge\vhoge")

>> hoge

hoge

hoge

**< input関数 >**

**input関数** を使うことでキーボードから値を入力して使用することができる.

print(input())

>> \_

()の中に指定した文字列は入力フィールドの前に表示される.

print(input('名前を入力してください : '))

>> 名前を入力してください : \_

試しに 山田太郎 と入力してEnterを押すと...

print(input('名前を入力してください : '))

>> 名前を入力してください : 山田太郎\_

山田太郎

**算術演算**

**< リテラル >**

**リテラル** ... プログラムに直接値を定義する定数のこと.  
整数や小数(浮動小数点数), 虚数, 文字列などが該当する.

**整数**  
※数字の間に\_を入れ, 外見上のみ桁をグループ化することができる.

10, 123456, 100\_000\_000

**浮動小数点数**

3.14, 3., .01, 3.14e-5, 3.14\_15\_92

**虚数**  
数字の最後尾にjを付けることで定義できる.

3.14j, 3j, .01j, 3.14e-5j, 3.14\_15\_92j

**文字列**  
' や " で囲うことで定義できる.

'apple', "apple", '円周率', '3'

**算術演算子**

**< 算術演算子 >**

**- 加算 -**

+演算子を用いることで数値リテラルの足し算が可能.

print(5 + 3)

>> 8

**- 減算 -**

-演算子を用いることで数値リテラルの引き算が可能.

print(5 - 3)

>> 2

**- 乗算 -**

\*演算子を用いることで数値リテラルの掛け算が可能.

print(5 \* 3)

>> 15

文字列に\*演算子を使用し, 数値を掛けることで文字列を繰り返す.

print("hoge" \* 2)

>> hogehoge

**- 乗累算 -**

\*\*演算子を用いることで数値リテラルの累乗算が可能.  
5∗∗3=53

print(5 \*\* 3)

>> 125

**- 除算 -**

/演算子を用いることで数値リテラルの割り算が可能.

print(5 / 3)

>> 1.6666666666666667

**- 切り捨て除算 -**

//演算子を用いることで小数点以下を切り捨てた商を求められる.

print(5 // 3)

>> 1

**- 剰余算 -**

%演算子を用いることで除算の余りを求められる.

print(5 % 3)

>> 2

**- 負数 -**

数値リテラルの前に-演算子

print(-3)

>> -3

**- 計算の優先順序 -**

原則、数学と同じ順序で計算される.

print(3 + 5\*2)

>> 13

()の中は優先的に計算される.

print((3+5) \* 2)

>> 16

**< 文字列演算 >**

+演算子を用いることで文字列同士の結合を行う．

print("hoge" + "hoge")

>> hogehoge

\*演算子を用いて，文字列に数値を掛けることで文字列を数値の回数繰り返す．

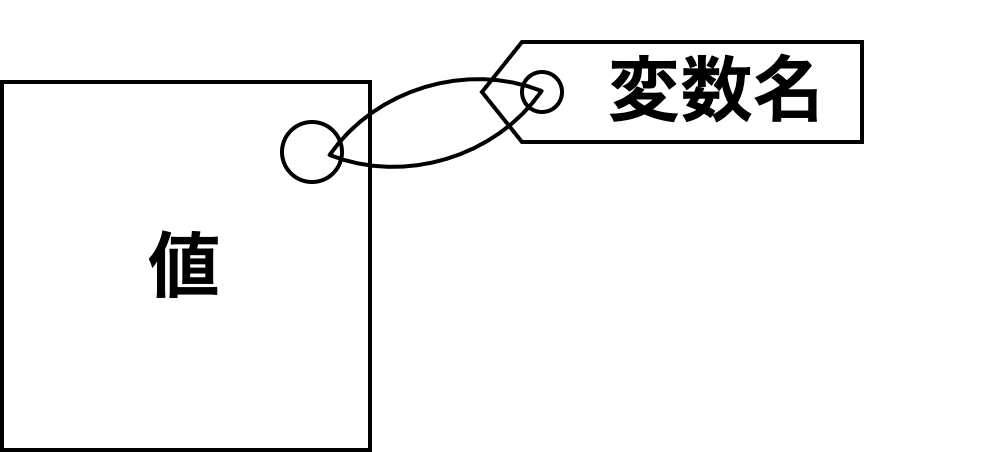
print("hoge" \* 2)

>> hogehoge

**変数**

**< 変数 >**

変数とは値にタグをつけるイメージ.  
値を直接使わずに変数を用いて演算を行うことができる．

[](https://camo.qiitausercontent.com/a6f6cb84c091f44549886ce2103bbab3496c6b35/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f31373237323736632d633535342d396466332d343064332d3863396635336132663764332e706e67)

**< 命名規則 >**

1. 使用できるのは **アルファベット** , **数字** , **\_(ｱﾝﾀﾞｰｽｺｱ)** の3種類. (日本語は原則NG)
2. 1文字目に **数字** は使えない.
3. 原則, 単語の区切りには **\_(ｱﾝﾀﾞｰｽｺｱ)** を使用する.
4. 原則, **変数** は小文字, **定数** は大文字のみで構成する.

**< 予約語 >**

**予約語** と呼ばれるPython内で指定されたキーワードは変数名に使えない.

【予約語】

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

**< 代入演算子 >**

**変数名 = 値** の形で変数を定義できる.  
この **=** を **代入演算子** と呼ぶ.

hoge = 5

print(hoge)

> 5

代入演算子では, 数学の等式とは意味が異なる.  
あくまで, 左辺の **変数** に 右辺の **値** を代入するためのものである.  
また, 変数の値は最後に代入した値を反映する.

hoge = 5

hoge = 10

print(hoge)

> 10

**< 変数の演算 >**

age = 19

age = age + 1

print(age)

> 20

1. 変数age に **19を代入** する ( age = 19 )
2. 変数age に **1を加算** する ( age + 1 )  
   -> 20
3. 変数age を **20に結びつける** ( age = 20 )

**< 複合演算子 >**

hoge = hoge + 1 は, hoge += 1 の形式に省略できる.  
この, **+=** の形を **複合演算子** と呼ぶ.  
+に限らず, 算術演算子であれば複合演算子としてまとめることが可能.

money = 2000

money -= 300

print(money)

>> 1700

**オブジェクト**

**< オブジェクト >**

**オブジェクト** ... **モノ**(データ)

Pythonのプログラムはすべて, **オブジェクト** もしくは **オブジェクト間の関係** である.  
すべてのオブジェクトは, **同一性(ID)**, **値**, **データ型** を持っている.

**< 同一性(ID) >**

**同一性(ID)** ... そのオブジェクトの **アドレス**

hoge = 1

print(id(1))

print(id("A"))

print(id(hoge))

>> 4421124336

4423945776

4421124336

**< データ型 >**

**データ型** ... そのデータの **性質**  
データ型によって, 取りうる値や可能な操作が決定する.  
データ型を調べる時は, **type関数** を使用する.

print(type(20))

print(type(20.5))

print(type('apple'))

>> <class 'int'>

<class 'float'>

<class 'str'>

**< キャスト(型変換) >**

あるオブジェクトのデータ型を別のデータ型に変換することを **キャスト** と呼ぶ.  
int(整数) -> str(文字列) に変換する場合は str(10) ,  
str(文字列) -> int(整数) に変換する場合は int('10')  
というようにそれぞれのデータ型に対応した関数を使用する.

integer = 10

print(type(integer))

string = str(integer)

print(type(string))

>> <class 'int'>

<class 'str'>

**数値型**

**< int >**

**int** とは, **整数値** を表すデータ型である.  
**整数リテラル** もしくは **int関数** を用いることで使用できる.  
また, 浮動小数点数にint関数を適用する場合は小数点以下を切り捨てる.

print(type(5))

print(type(int('5')))

print(type(int(2.8)))

>> <class 'int'>

<class 'int'>

<class 'int'>

**< float >**

**float** とは, **小数** を表すデータ型である.

print(type(3.14))

print(type(float('3.14')))

print(type(2e-3))

>> <class 'float'>

<class 'float'>

<class 'float'>

**< complex >**

**complex** とは, **複素数** を表すデータ型である.

print(type(5j))

print(type(3+5j))

>> <class 'complex'>

<class 'complex'>

**文字列**

**< str >**

**str** とは， 文字列を表すデータ型である．

文字列リテラルもしくはstr関数を用いることで使用できる

print("Hello", type("Hello"))

print(str(5), type(str(5))

>> Hello <class 'str'>

5 <class 'str'>

**文字列の置換**

**replaceメソッド** を使うことで文字列の一部を **置換** することができる.

文字列.replace(置換前, 置換後)

name = "Yamada Taro"

replace\_name = name.replace("Taro", "Jiro")

print(replace\_name)

>> Yamada Jiro

**文字列の所属関係**

**startswithメソッド** を使うことでその文字列が指定した文字列で始まるかを判定する．

print("Hello World".startswith("Hello")

>> True

**endswithメソッド** を使うことでその文字列が指定した文字列で終わるかを判定する．

print("Hello World".endswith("World")

>> True

**文字列の位置関係**

**findメソッド** はその文字列のうち， 指定した文字列が前方から何文字目に位置するかを返す．

print("Hello World".find("o")

>> 4

**rfindメソッド** も同じく， その文字列のうち， 指定した文字列が前方から何文字目に位置するかを返す．

findメソッドはヒットした文字列のうち最前方のインデックスを返すのに対して，

rfindメソッドは最後方のインデックスを返す．

print("Hello World".rfind("o")

>> 7

また， どちらも存在しない場合は-1を返す．

print("Hello World".find("P"))

>> -1

同じ機能を持つが， 存在しない場合にValueErrorを返すメソッドとして  
indexメソッド, rindexメソッドがある．

print("Hello World".index("o"))

print("Hello World".rindex("o"))

>> 4

7

print("Hello World".index("P"))

>> ValueError: substring not found

**文字列の構成要**

**isalnumメソッド** はその文字列が  
アルファベットまたは数字のみで構成されていればTrue, そうでなければFalseを返す

print("Hello2023".isalnum())

>> True

**isalphaメソッド** はその文字列が  
アルファベットのみで構成されていればTrue, そうでなければFalseを返す．

print("HelloWorld".isalpha())

>> True

**isdigitメソッド** はその文字列が  
数字のみで構成されていればTrue, そうでなければFalseを返す．

print("2023".isdigit())

>> True

**islowerメソッド** はその文字列に含まれるアルファベットが  
小文字のみで構成されていればTrue, そうでなければFalseを返す．  
アルファベット以外は無視される．

print("hello world".islower())

>> True

**isupperメソッド** はその文字列に含まれるアルファベットが  
大文字のみで構成されていればTrue, そうでなければFalseを返す．

アルファベット以外は無視される．

print("HELLO WORLD".isupper())

>> True

**isspaceメソッド** はその文字列が  
空白のみで構成されていればTrue, そうでなければFalseを返す．

print(" ".isspace())

>> True

**istitleメソッド** はその文字列に含まれる単語が

すべて1文字目のみが大文字であればTrue, そうでなければFalseを返す．

print("Hello World".istitle())

>> True

**文字列の変換**

**capitalizeメソッド** はその文字列の先頭の一文字を大文字， 他を小文字に変換する．

print("HELLO WORLD".capitalize())

>> Hello world

**titleメソッド** はその文字列の単語の先頭一文字を大文字， 他を小文字に変換する.

print("HELLO WORLD".title())

>> Hello World

**upperメソッド** はその文字列の小文字を大文字に変換する.

print("hello world".upper())

>> HELLO WORLD

**lowerメソッド** はその文字列の大文字を小文字に変換する.

print("HELLO WORLD".lower())

>> hello world

**swapcaseメソッド** はその文字列の大文字を小文字， 小文字を大文字に変換する.

print("Hello World".swapcase())

>> hELLO wORLD

**文字列の分割**

**splitメソッド** はその文字列を指定した文字列で分割する．

print("apple, banana, lemon".split(','))

print("Hello World".split())

>> ['apple', ' banana', ' lemon']

['Hello', 'World']

引数を省略した場合は空白文字で分割される．  
**splitlinesメソッド** はその文字列を改行コードで分割する．

print("Hello\nWorld".splitlines())

>> ['Hello', 'World']

**シーケンス**

**< シーケンス >**

**シーケンス** とは, **複数の要素を格納したデータ構造** の総称である.  
また, **要素を1つずつ取り出す** ことのできる **イテラブルオブジェクト** の一種である.

後ほど詳しく説明するが, シーケンスの要素を指定する方法には  
単一要素を対象とする **index** と, 複数要素を対象とする **slice** が存在する.  
[](https://camo.qiitausercontent.com/f917c0909386d4b7229c56b22c585938a184924b/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f36346261356563352d626133652d383066622d343864352d3436393064346636656534372e706e67)

**< list >**

代表的なシーケンスに **リスト** がある.  
リストは定義した後に **要素の変更が可能** な **ミュータブル** なオブジェクトである.  
**リスト** は [要素1, 要素2, 要素3] のように []の中を **,(ｶﾝﾏ)** で区切る.

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

print(fruits)

>> ['apple', 'banana', 'lemon']

**- index -**

シーケンスでは, list[0] のように **index(見出し番号)** を指定して取り出すことができる.  
**index** は1番目からではなく, **0番目** からであることに注意.

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

print(fruits[0])

print(fruits[1])

print(fruits[2])

>> apple

banana

lemon

list[-1] のように **index(見出し番号)** を  
**-(ﾏｲﾅｽ)** で指定することで要素を後ろから取り出すことができる.  
[-1] で **最後** の要素を, [-2] で **後ろから2番目** の要素を取り出すことができる.

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

print(fruits[-1])

print(fruits[-2])

print(fruits[-3])

>> lemon

banana

apple

**- slice -**

シーケンスでは, **slice記法** を用いることで **範囲を指定して** 要素を取り出すことができる.  
**slice記法 : [最初の要素のindex : 最後の要素のindex+1]**

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

print(fruits[0:2])

print(fruits[1:3])

print(fruits[0:1])

>> ['apple', 'banana']

['banana', 'lemon']

['apple']

slice記法では, 一方を省略することが可能である.  
[: 最後の要素のindex+1] とすることで最初の要素から指定したindexまでを,  
[最初の要素のindex :] とすることで指定したindexから最後の要素までを指定できる,

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

print(fruits[:2])

print(fruits[1:])

>> ['apple', 'banana']

['banana', 'lemon']

**- step -**

**slice記法** には始点と終点の他に **ステップ** と呼ばれる **増分** パラメータを指定できる.  
デフォルトでは1が指定されている.  
**slice記法 : [最初の要素のindex : 最後の要素のindex+1 : 増分]**

hoge = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(hoge[0:6:2])

print(hoge[1::2])

print(hoge[::3])

>> [1, 3, 5]

[2, 4, 6, 8, 10]

[1, 4, 7, 10]

**- 要素の更新 -**

**index** を指定して値を代入することでリストを **更新** することができる.

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

fruits[0] = 'melon'

print(fruits)

>> ['melon', 'banana', 'lemon']

**slice** で指定しても同じくリストを **更新** することができる.

fruits[1:3] = ['orange', 'strawberry']

print(fruits)

>> ['melon', 'orange', 'strawberry']

**- 要素の追加 -**

**appendメソッド** を使うことでリストの **最後尾** に要素を **追加** することができる.  
**list.append(data)**

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

fruits.append('orange')

print(fruits)

>> ['apple', 'banana', 'lemon', 'orange']

**- 要素の拡張 -**

**extendメソッド** を使うことでリストの **最後尾** に **他のリストの要素を追加** することができる.  
**list.extend(list)**

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

fruits\_2 = ['orange', 'melon']

fruits.extend(fruits\_2)

print(fruits)

>> ['apple', 'banana', 'lemon', 'orange', 'melon']

**- 要素の挿入 -**

**insertメソッド** を使うことでリストの **指定した位置** に要素を **挿入** することができる.  
指定する位置は **slice視点のindex** であることに注意.  
**list.insert(index, data)**

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

fruits.insert(2, 'lemon')

print(fruits)

>> ['apple', 'banana', 'lemon', 'orange']

**- 要素の削除 -**

**del文** を使うことでリストの **指定した位置** の要素を **削除** することができる.  
**del list[index]**

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

del fruits[1]

print(fruits)

>> ['apple', 'lemon']

**removeメソッド** を使うことでリストの **指定した値** を **削除** することができる.  
ただし, 該当する最初の要素のみを削除する.  
**list.remove(data)**

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

fruits.remove('banana')

print(fruits)

>> ['apple', 'lemon']

**popメソッド** を使うことでリストの **指定した位置** の要素を **削除** し, その値を取得する.  
**list.pop(index)**

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

fruits.pop(1)

print(fruits)

>> ['apple', 'lemon']

**clearメソッド** を使うことでリストの **すべての要素** を **削除** する.  
**list.clear()**

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

fruits.clear()

print(fruits)

>> []

**- 要素の並び替え -**

**sortメソッド** を使うことでリストの要素を **昇順に並び替える** ことができる.  
また, 引数にreverse=True を指定することで **降順に並び替える** ことができる.  
**昇順 : list.sort()**       **降順 : list.sort(reverse=True)**

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

hoge.sort()

print(hoge)

hoge.sort(reverse=True)

print(hoge)

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

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

sortメソッドではリストを直接書き換えていたが,  
**sorted関数** を使うと **元のリストは残しつつ**, 昇順に並び替えたリストを返すことができる.  
sorted関数でも第二引数にreverse=True を指定することで **降順** にすることができる.

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

fuga = sorted(hoge, reverse=True)

print(hoge)

print(fuga)

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

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

**- 要素の検索 -**

**indexメソッド** を使うことでその要素がリストの **何番目に位置するか** 検索することができる.  
ただし, 該当する最初のインデックスのみを返す.  
**list.index(data)**

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

print(fruits.index('banana'))

>> 1

**- 要素のカウント -**

**countメソッド** を使うことでその要素がリストに **何個含まれているか** 計数することができる.  
**list.count(data)**

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

print(fruits.count('apple'))

>> 3

**- 要素数 -**

**len関数** を使うことでそのイテラブルオブジェクトの **要素数** を求めることができる.  
**len(リスト)**

hoge = [1,2,3,4,5,6,7,8,9,10]

print(len(hoge))

>> 10

**- 合計値 -**

**sum関数** を使うことでそのイテラブルオブジェクトの **合計値** を求めることができる.  
**sum(リスト)**

hoge = [1,2,3,4,5,6,7,8,9,10]

print(sum(hoge))

>> 55

**- 最大値 -**

**max関数** を使うことでそのイテラブルオブジェクトの **最大値** を求めることができる.  
**max(リスト)**

hoge = [1,2,3,4,5,6,7,8,9,10]

print(max(hoge))

>> 10

**- 最小値 -**

**min関数** を使うことでそのイテラブルオブジェクトの **最小値** を求めることができる.  
**min(リスト)**

hoge = [1,2,3,4,5,6,7,8,9,10]

print(min(hoge))

>> 1

**- 二次元配列 -**

Python上でExcelのような縦横の表を作成したい場合は,  
リストの中にリストを並べることで表現する.

|  | **A列** | **B列** | **C列** |
| --- | --- | --- | --- |
| **1行** | A1 | B1 | C1 |
| **2行** | A2 | B2 | C2 |
| **3行** | A3 | B3 | C3 |

hoge = [['A1', 'B1', 'C1'],

['A2', 'B2', 'C2'],

['A3', 'B3', 'C3']]

二次元配列では, list[row][column] のように行と列を指定して取り出すことができる.

hoge = [['A1', 'B1', 'C1'],

['A2', 'B2', 'C2'],

['A3', 'B3', 'C3']]

print(hoge[0])

print(hoge[0][1])

>> ['A1', 'B1', 'C1']

B1

**< タプル >**

リストによく似たデータ型に **タプル** がある.  
タプルは定義した後に **要素の変更が不可能** な **イミュータブル** なオブジェクトである.  
**タプル** は (要素1, 要素2, 要素3) のように () の中を **,(ｶﾝﾏ)** で区切る.

fruits = ('apple', 'banana', 'lemon')

fruits[0] = 'melon'

>> TypeError: 'tuple' object does not support item assignment

エラー内容 : tupleは要素の割り当てができない.

**- rangeオブジェクト -**

**rangeオブジェクト** は **連続する整数** を要素とするな **イミュータブル** なオブジェクトである.  
**range(最小値, 最大値+1, 増分)** (最小値と増分は省略可能)  
rangeオブジェクトの要素を確認するには, 一度リストなどに変換する必要がある.

print(range(0,10,1))

print(list(range(0,10,1)))

print(list(range(10)))

print(list(range(0,10, 2)))

>> range(0, 10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[0, 2, 4, 6, 8]

**- テキストシーケンス -**

文字列もイミュータブルなシーケンスの一種である.

print("apple"[0])

print("apple"[0:3])

print(len("apple"))

>> a

app

5

**マッピング**

**< dictionary >**

**マッピング** とは, 検索に用いる **key** と 値である **value** をまとめて格納するデータ型である.  
Pythonでは **辞書** がこれに該当し, **イテラブルオブジェクト** の一種である.  
{key1: value1, key2: value2} のように {} の中に key: value の形で格納する.

fruits = {'apple': 180, 'banana': 100, 'lemon': 85}

print(fruits)

>> {'apple': 180, 'banana': 100, 'lemon': 85}

**- dict関数 -**

前述した {key1: value1, key2: value2} 以外にも **dict関数** を使用する定義方法がある.

1. dict(key1=value1, key2=value2)  
   (このとき, keyに文字列を指定する場合はクォーテーションが不要であることに注意)
2. dict([(key1, value1), (key2, value2)])

fruits\_1 = dict(apple=180, banana=100, lemon=85)

fruits\_2 = dict([('melon', 1000), ('orange', 130)])

print(fruits\_1)

print(fruits\_2)

>> {'apple': 180, 'banana': 100, 'lemon': 85}

{'melon': 1000, 'orange': 130}

2つのシーケンスから辞書を生成する場合は **zip関数** を使用する.  
3. keys = [key1, key2]  
  values = [value1, value2]  
  dict(zip(keys, values))

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

prices = [180, 100, 85]

fruits\_and\_prices = dict(zip(fruits, prices))

print(fruits\_and\_prices)

>> {'apple': 180, 'banana': 100, 'lemon': 85}

**- keyからvalueの取得 -**

dict[key] のように **key** を指定することで対応する **value** を取得できる.  
もしくは, **getメソッド** を使用し, dict.get(key) と指定することもできる.

fruits = {'apple': 180, 'banana': 100, 'lemon': 85}

print(fruits['apple'])

print(fruits.get('apple'))

>> 180

180

**- KeyError -**

dict[key] 記法では, 存在しないkeyを指定すると **KeyError** となる.

print(fruits['orange'])

>> KeyError: 'orange'

dict.get(key) 記法では, 存在しないkeyを指定すると **None** を返す.  
また, 第二引数に存在しないkeyを指定した場合の戻り値を指定することもできる.

print(fruits.get('orange'))

print(fruits.get('orange', '存在しない'))

>> None

存在しない

**- key/valueの一覧を取得 -**

下記のメソッドを使用することで **イミュータブル** な **ビューオブジェクト** として取得する.  
**keys :** dict.keys()  
**values :** dict.values()  
**keys & values :** dict.items()

fruits = {'apple': 180, 'banana': 100, 'lemon': 85}

print(fruits.keys())

print(fruits.values())

print(fruits.items())

>> dict\_keys(['apple', 'banana', 'lemon'])

dict\_values([180, 100, 85])

dict\_items([('apple', 180), ('banana', 100), ('lemon', 85)])

**- keyの有無 -**

**keysメソッド** と **in演算子** を組み合わせることで **そのkeyを含むか** 検索できる.  
key in dict.keys()  
また, keysメソッドを省略して key in dict と書くこともできる.

fruits = {'apple': 180, 'banana': 100, 'lemon': 85}

print('apple' in fruits)

print('orange' in fruits)

>> True

False

**- valueの有無 -**

**valuesメソッド** と **in演算子** を組み合わせることで **そのvalueを含むか** 検索することができる.  
value in dict.values()

fruits = {'apple': 180, 'banana': 100, 'lemon': 85}

print(100 in fruits.values())

print(200 in fruits.values())

>> True

False

**- key & valueの有無 -**

**itemsメソッド** と **in演算子** を組み合わせることで **そのkey: valueを含むか** 検索できる.  
(key, value) in dict.items()

fruits = {'apple': 180, 'banana': 100, 'lemon': 85}

print(('apple', 180) in fruits.items())

print(('apple', 200) in fruits.items())

>> True

False

**- 辞書を用いたfor文 -**

itemsメソッド を使用することで **key** と **value** を同時に取り出すことができる.

fruits = {'apple': 180, 'banana': 100, 'lemon': 85}

for key, value in fruits.items():

print(key, value)

>> apple 180

banana 100

lemon 85

**集合**

**- set -**

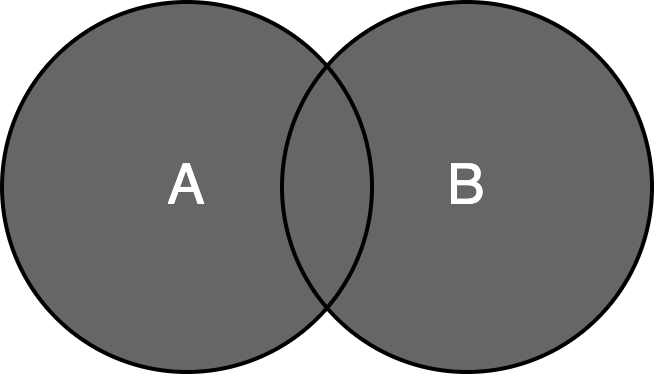
**集合** を扱うためのシーケンスに **セット** がある.  
セットは **ユニークな要素** を持つ **ミュータブル** なオブジェクトである.  
また, セットでは要素の **順番を保持しない** ため, インデックスによる指定等ができない.  
セットは {要素1, 要素2, 要素3} のように {} の中を **,(ｶﾝﾏ)** で区切る.

fruits = {'apple', 'apple', 'apple', 'banana', 'banana', 'lemon'}

print(fruits)

>> {'lemon', 'apple', 'banana'}

**- 和集合 -**

**和集合** を求める場合は **|演算子** を使用する.  
[](https://camo.qiitausercontent.com/be08e91341474c21e08560fc3f5188c6fdad9fb3/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f61366365633465362d643233372d363262382d633637642d3463393462383664376565312e706e67)

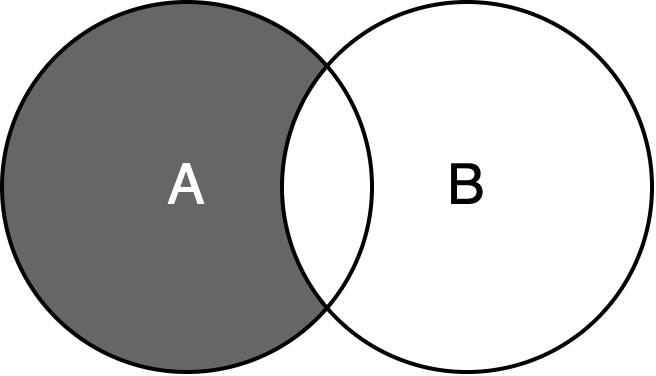
A = {2, 4, 6, 8, 10, 12}

B = {3, 6, 9, 12, 15, 18}

print(A | B)

>> {2, 3, 4, 6, 8, 9, 10, 12, 15, 18}

**- 差集合 -**

**差集合** を求める場合は **-演算子** を使用する.  
[](https://camo.qiitausercontent.com/a864ade9285df2455cd54be9d1af2e7d523f5b6b/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f31343737343839372d646164352d333963332d356439342d6162353234653265343233352e706e67)

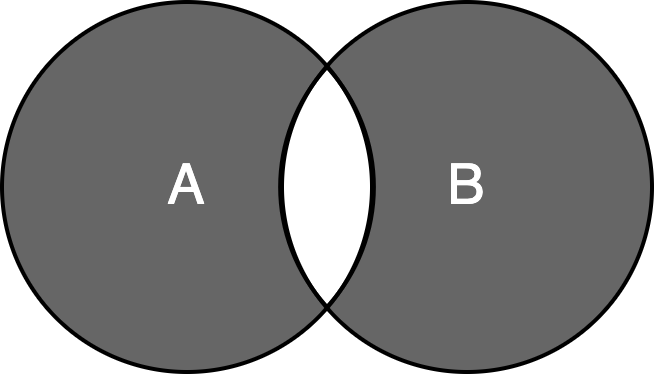
A = {2, 4, 6, 8, 10, 12}

B = {3, 6, 9, 12, 15, 18}

print(A - B)

>> {8, 2, 10, 4}

**- 対称差 -**

**対称差** を求める場合は **^演算子** を使用する.  
[](https://camo.qiitausercontent.com/be452d691f4e0f53fb2873d2ae4908dfb103f2b7/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f62366363646636392d396534352d646264322d313761612d3531343563373763613466642e706e67)

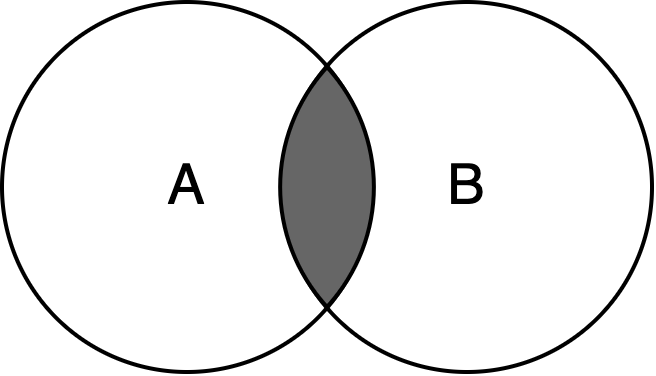
A = {2, 4, 6, 8, 10, 12}

B = {3, 6, 9, 12, 15, 18}

print(A ^ B)

>> {2, 3, 4, 8, 9, 10, 15, 18}

**- 積集合 -**

**積集合** を求める場合は **&演算子** を使用する.  
[](https://camo.qiitausercontent.com/338b720cf4c25f1860668564522d1dc3abfc12f0/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f38393163393837632d633838362d346263622d306661612d3532323233626238303437302e706e67)

A = {2, 4, 6, 8, 10, 12}

B = {3, 6, 9, 12, 15, 18}

print(A & B)

>> {12, 6}

**- 要素の追加 -**

**addメソッド** を使うことでセットに要素を **追加** することができる.

fruits = {'apple', 'banana', 'lemon'}

fruits.add('orange')

print(fruits)

>> {'lemon', 'orange', 'apple', 'banana'}

**- 要素の削除 -**

**removeメソッド** を使うことでセットから指定した要素を **削除** することができる.  
ただし, removeメソッドでは **存在しないkey** を指定すると **KeyError** となる.  
代わりに, **discardメソッド** を使用することでKeyErrorを回避することができる.

fruits = {'apple', 'banana', 'lemon'}

fruits.remove('banana')

fruits.discard('orange')

print(fruits)

>> {'lemon', 'apple'}

**popメソッド** を使うことでセットから **いずれかの要素** を **削除** し, その値を取得する.  
削除する要素を指定することはできない.

fruits = {'apple', 'banana', 'lemon'}

print(fruits.pop())

print(fruits)

>> lemon

{'apple', 'banana'}

**clearメソッド** を使うことでセットの **すべての要素** を **削除** する.

fruits = {'apple', 'banana', 'lemon'}

fruits.clear()

print(fruits)

>> set()

**- イミュータブルなセット -**

イミュータブルなセットとして **frozensetオブジェクト** がある.  
frozensetは frozenset({要素1, 要素2, 要素3}) のように **frozenset関数** を使用する.

fruits = frozenset({'apple', 'banana', 'lemon'})

fruits.add('orange')

>> AttributeError: 'frozenset' object has no attribute 'add'

エラー内容 : frozensetオブジェクトではaddメソッドを使用できない.

**条件分岐**

**< 真偽値 >**

ある条件式が成り立つ場合は **True** , 成り立たない場合は **False** を返す．  
この True, False は **bool** と呼ばれるデータ型である.

print(5 > 3)

print(5 < 3)

print(type(True))

>> True

False

<class 'bool'>

boolはintのサブクラスであり,  
Trueは1を, Falseは0の値を取る.

print(int(True))

print(True + True)

>> 1

2

**< 比較演算子 >**

**- 等価 / 不等価 -**

A == B : AとBが **等しい** のであれば True , そうでなければ False を返す.  
A != B : AとBが **等しくない** のであれば True , そうでなければ False を返す.

hoge = 5

print(hoge == 5)

print(hoge != 5)

>> True

False

**- 大なり / 小なり -#**

A > B : AがB **より大きい** ならば True , そうでなければ False を返す.  
A < B : AがB **より小さい** ならば True , そうでなければ False を返す.

hoge = 5

print(hoge > 4)

print(hoge > 5)

print(hoge > 6)

>> True

False

False

**- 以上 / 以下 -**

A >= B : AがB **以上** であれば True , そうでなければ False を返す.  
A <= B : AがB **以下** であれば True , そうでなければ False を返す.

hoge = 5

print(hoge >= 4)

print(hoge >= 5)

print(hoge >= 6)

>> True

True

False

**< 所属検査演算子 >**

**- 含む -**

A in B : AがBに **含まれている** のであれば True , そうでなければ False を返す.  
Bには **リスト** や **文字列** などを使用することができる.

print('apple' in ['apple', 'banana', 'lemon'] )

print('melon' in ['apple', 'banana', 'lemon'] )

print('a' in 'apple')

>> True

False

True

**< 論理演算子 >**

**- 論理積 -**

A and B : A **かつ** B であれば True , そうでなければ False を返す.  
        (AとBのどちらとも **True** であれば, **True**)

print(True and True)

print(True and False)

print(False and False)

>> True

False

False

**- 論理和 -**

A or B : A **または** B であれば True , そうでなければ False を返す.  
        (AかBのどちらか一方が **True** であれば, **True**)

print(True or True)

print(True or False)

print(False or False)

>> True

True

False

**- 否定 -**

not A : True と False が反転する.

print(not True)

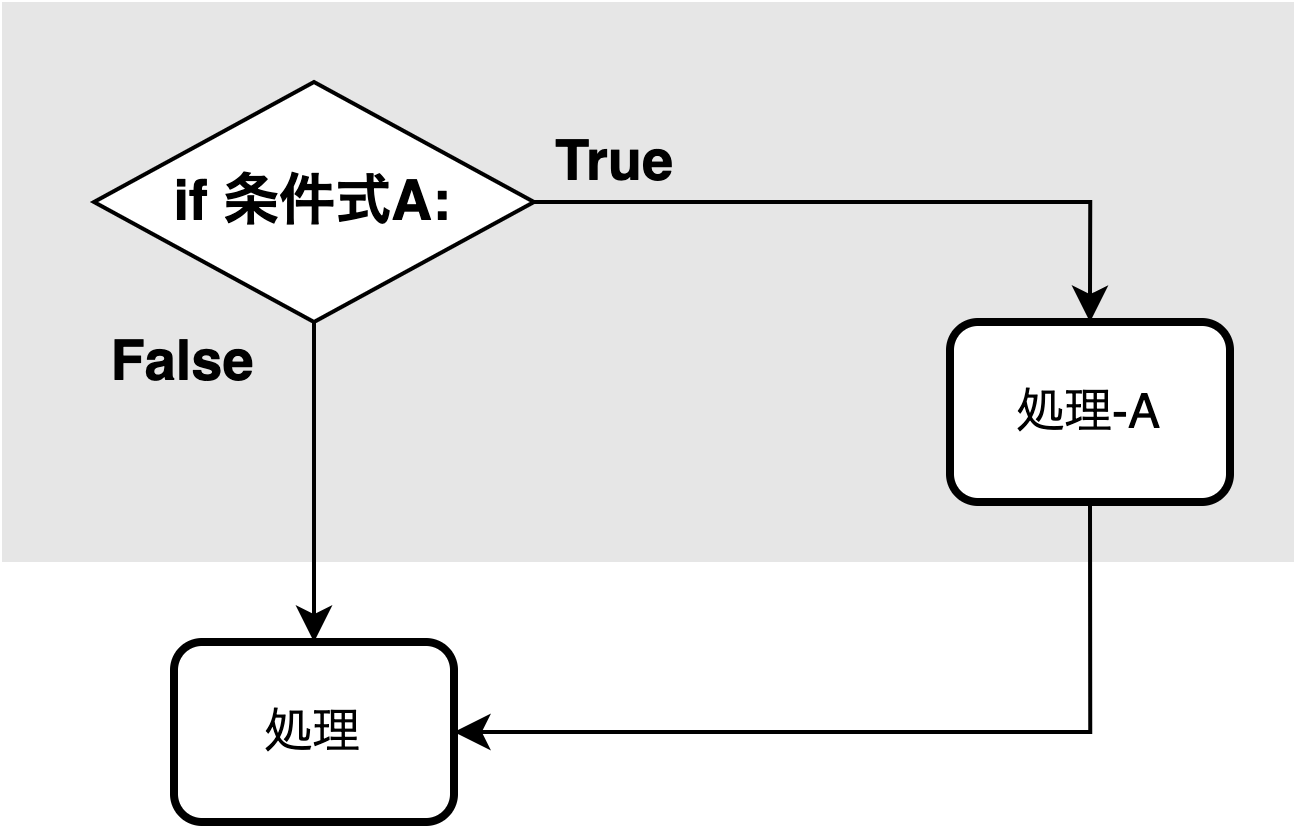
print(not False)

>> False

True

**< if文 >**

これまで紹介してきたプログラムは必ず **上から全て実行** された. これを **順次処理** と呼ぶ.  
これに対して, 条件に従って **処理の内容をわける** ものを **条件分岐処理** と呼ぶ.

ある条件に当てはまった場合の処理を指定したい場合は **if文** を使う.  
[](https://camo.qiitausercontent.com/feaa0c7566aa22ef216f298c9031b4cce6bb3cf5/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f37616263313635392d383663382d396565372d623931622d3861613264623464646665322e706e67)

**if [条件式]:** と書くことでその後の処理は **条件式がTrueの場合のみ** 実行される.  
if文の適用範囲は **インデント(原則スペース4つ)** を下げることで表現する.

weather = "晴れ"

if weather == "晴れ":

print('晴れのため, 傘は不要です.')

if weather == "雨":

print('雨のため, 傘が必要です.')

print('いってらっしゃいませ')

>> 晴れのため, 傘は不要です.

いってらっしゃいませ

if文の判定は独立して行われるため, if文を並べると条件に当てはまる全ての処理が実行される.

hoge = 8

if hoge >= 10:

print("hogeは10以上です")

if hoge >= 5:

print("hogeは5以上です")

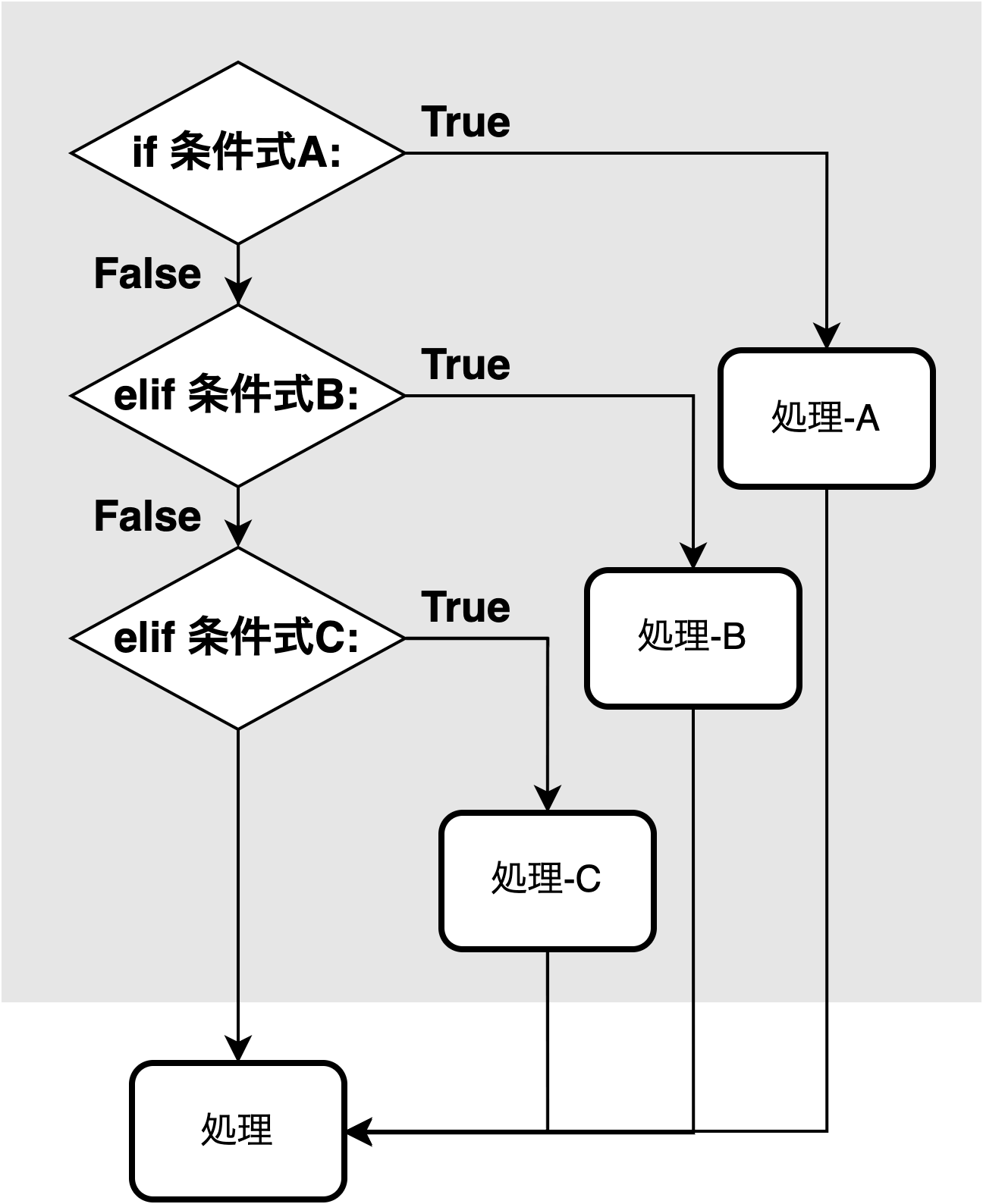
if hoge >= 0:

print("hogeは0以上です")

>> hogeは5以上です

hogeは0以上です

**- elif文 -**

複数の条件のうち, 最初に当てはまった条件の処理のみを実行したい場合は **elif文** を使う.  
[](https://camo.qiitausercontent.com/fd2aa0f845ccf7c2049bace5710473899ebbfe9f/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f31316334383934622d353162382d353133302d623138652d3630376539333665623761652e706e67)

**elif [条件式]:** と書くことでその後の処理は **条件式がTrueの場合のみ** 実行される.  
if文の適用範囲は **インデント(原則スペース4つ)** を下げることで表現する.

hoge = 8

if hoge >= 10:

print("hogeは10以上です")

elif hoge >= 5:

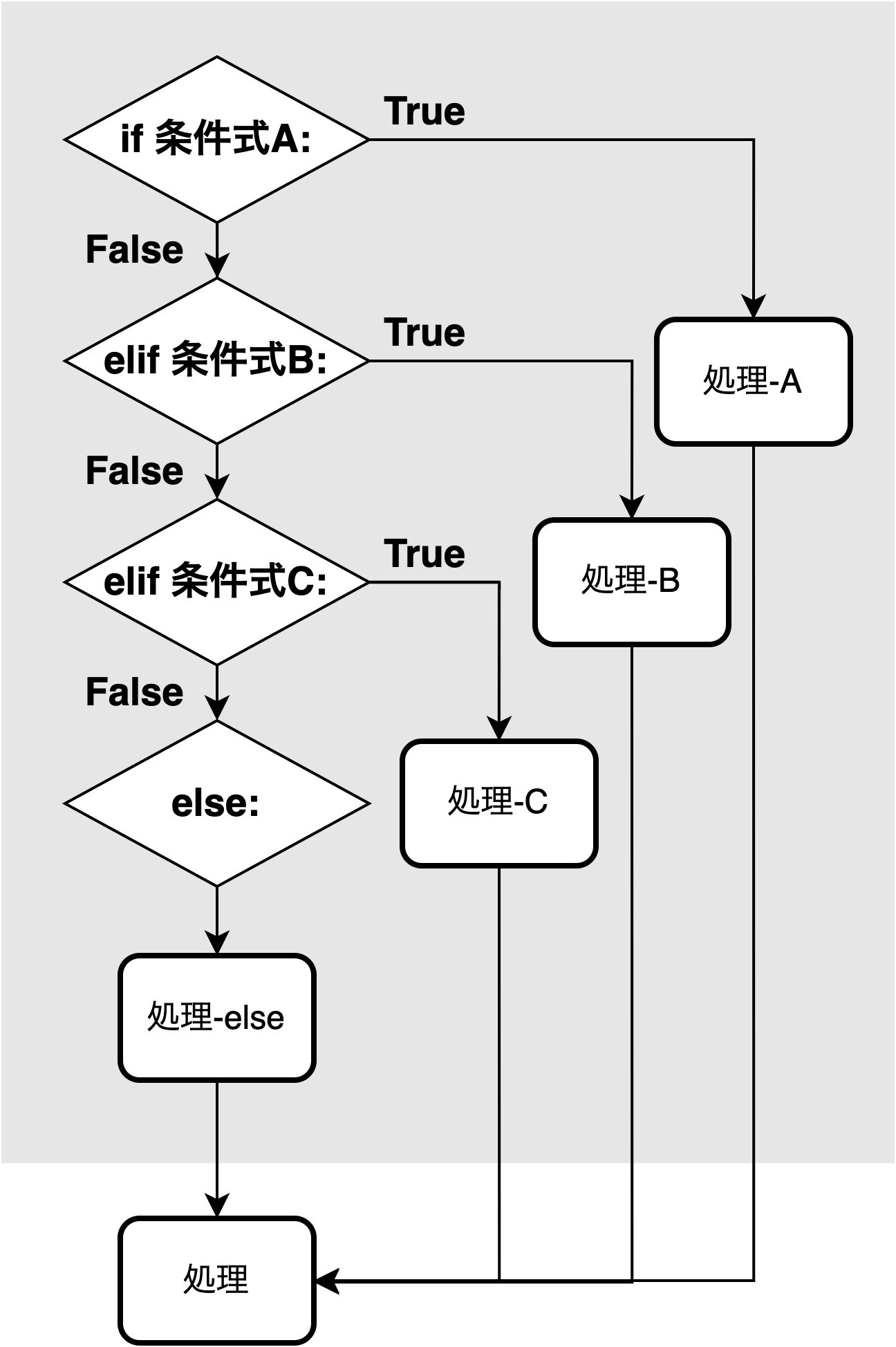
print("hogeは5以上です")

elif hoge >= 0:

print("hogeは0以上です")

>> hogeは5以上です

**< else文 >**

どの条件にも当てはまらなかった場合の処理を指定したい場合は **else文** を使う.  
[](https://camo.qiitausercontent.com/e19d1240de6344bffd6cbe7c94001a10d47f72ce/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f30346166616230322d333365352d613137332d613933652d3262313965323435373338332e706e67)

**else:** と書くことでその後の処理は **if/elif文のどれにも当てはまらない場合のみ** 実行される.

hoge = -5

if hoge >= 10:

print("hogeは10以上です")

elif hoge >= 5:

print("hogeは5以上です")

elif hoge >= 0:

print("hogeは0以上です")

else:

print("hogeは負数です")

>> hogeは負数です

**< 三項演算子 >**

**三項演算子** を用いることで **if文** による条件分岐を **1行** で記述することができる.  
ただし, 三項演算子を使うことでコードが複雑になる可能性があるため乱用には注意.  
[True処理] if [条件式] else [False処理]

hoge = -5

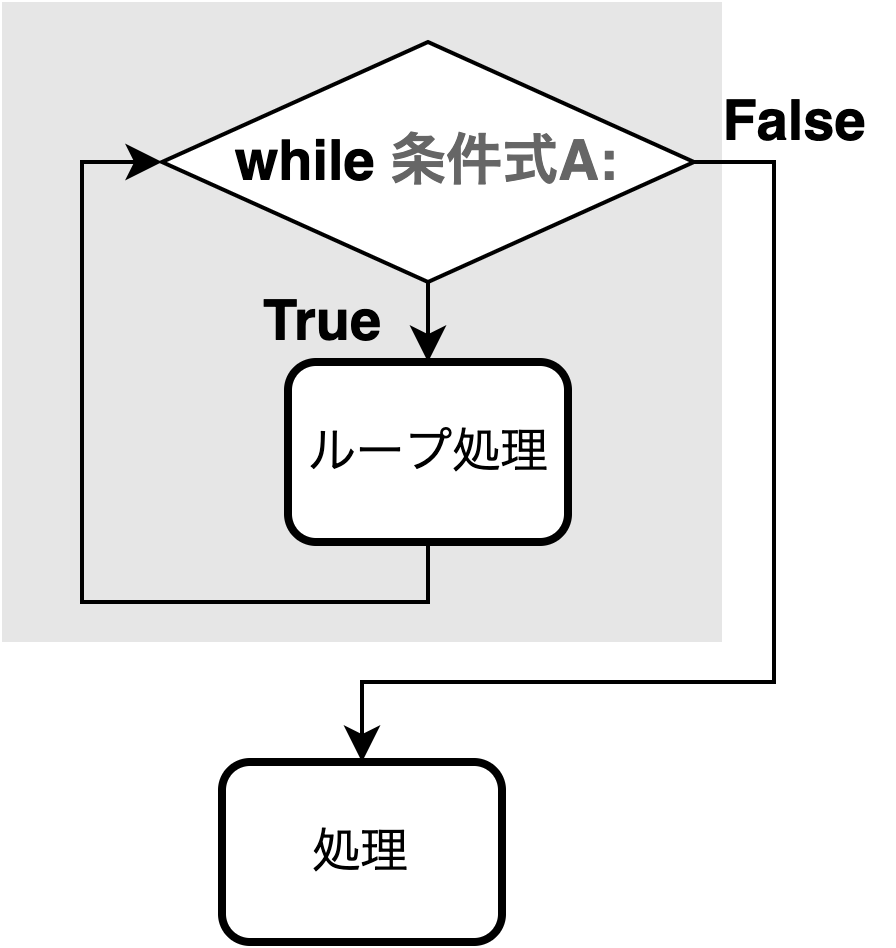
print("hogeは0以上です") if hoge >= 0 else print("hogeは負数です")

>> hogeは負数です

**繰り返し**

**< while文 >**

ある処理を100回繰り返す場合、100行そのプログラムを記述する必要がある.  
そこで, Pythonを含む多くのプログラミング言語では **繰り返し処理** が存在する.

ある条件に当てはまっている間, 処理を繰り返したい場合は **while文** を使う.  
[](https://camo.qiitausercontent.com/de29baa5856b860fb914b21f29dad02cd4f179ed/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f32363534316636612d393466622d343332312d363232352d6265323731333162386561342e706e67)

**while [条件式]:** と書くことでその後の処理は **条件式がTrueの間ずっと** 実行される.  
while文の適用範囲は **インデント(原則スペース4つ)** を下げることで表現する.

hoge = 1

while hoge <= 50:

hoge \*= 2

print(hoge)

print("プログラムを終了します")

>> 2

4

8

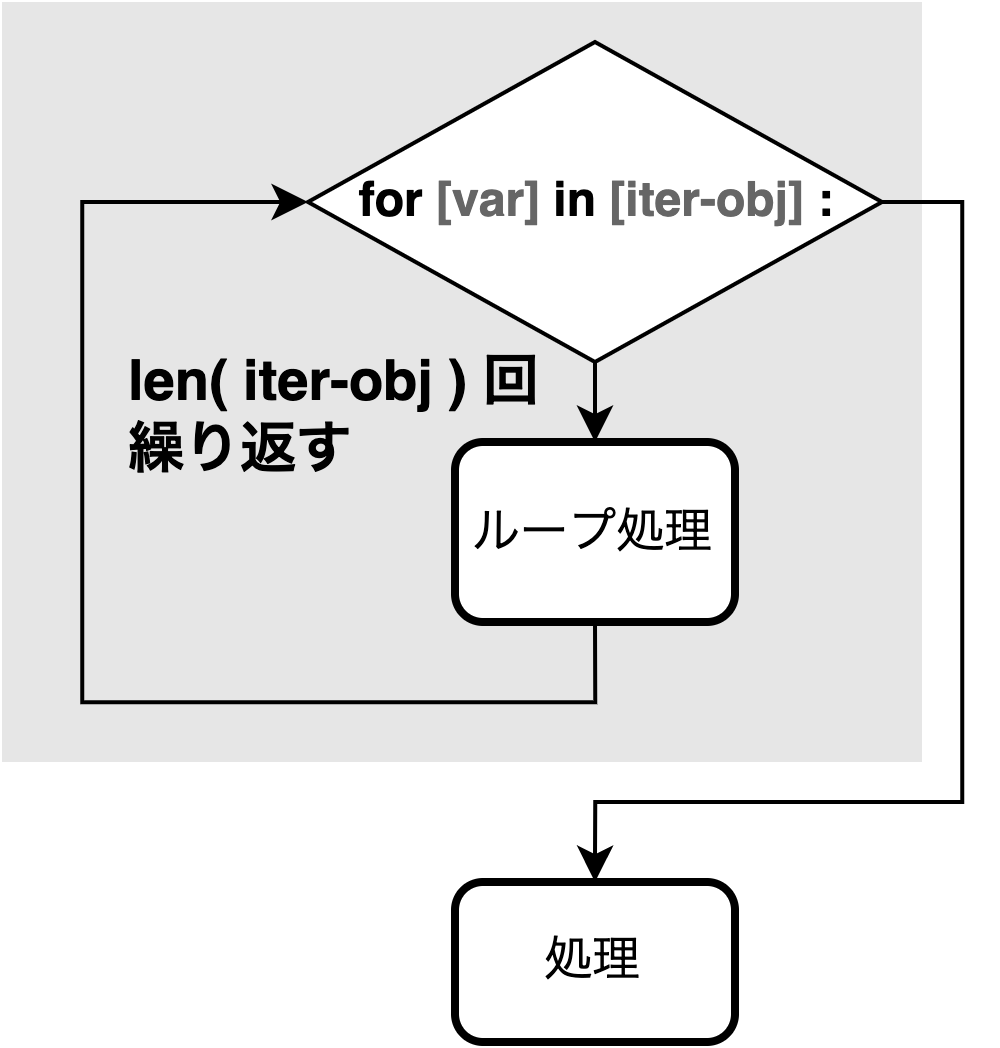
16

32

64

プログラムを終了します

**< for文 >**

特定の回数だけ処理を繰り返したい場合は **for文** を使う.  
[](https://camo.qiitausercontent.com/c56d77201b722b07f5df3524989d0978bc2d51aa/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f35333439616262352d396661392d656561332d303136392d3133666334626331363037322e706e67)

**for [変数] in [イテラブルオブジェクト]:** と書くことで  
イテラブルオブジェクトの **要素を1つずつ** 取り出し, **変数に格納** する.  
イテラブルオブジェクトの最後の要素に到達したのち, 繰り返し処理を終了する.

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

for fruit in fruits:

print(fruit)

print("プログラムを終了します")

>> apple

banana

lemon

プログラムを終了します

回数でしたい場合は **rangeオブジェクト** を使用する.

for i in range(5):

print(i)

print("プログラムを終了します")

>> 0

1

2

3

4

プログラムを終了します

後ろの要素から取得したい場合は **reversed関数** を使用する.

for i in reversed(range(5)):

print(i)

print("プログラムを終了します")

>> 4

3

2

1

0

プログラムを終了します

要素とインデックスを同時に取得したい場合は **enumerate関数** を使用する.

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

for i, fruit in enumerate(fruits):

print(i, fruit)

print("プログラムを終了します")

>> 0 apple

1 banana

2 lemon

プログラムを終了します

複数のイテラブルオブジェクトを扱いたい場合は **zip関数** を使用する.

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

prices = [180, 100, 85]

for fruit, price in zip(fruits, prices):

print(fruit, price)

print("プログラムを終了します")

>> apple 180

banana 100

lemon 85

プログラムを終了します

**関数**

**< 関数 >**

**関数** ... 処理を定義し，再利用を可能とする制御構造のこと．

Pythonにあらかじめ用意されている **組み込み関数** と，  
自分で定義することで使用できる **ユーザ定義関数** の2種類がある．

**< def文 >**

関数を定義するときは **def文** を使用する.  
def 関数名(): で定義し， 関数名()で呼び出すことができる.

def greeting():

print("Hello world!")

greeting()

>> Hello world!

**< 引数 >**

関数を呼び出す際に，()に値を渡すことで関数内で使用することができる.  
関数側で定義する引数を **仮引数** ，  
呼び出し時に渡す引数を **実引数** と呼ぶ．

def greeting\_name(name):

print("Hello " + name + "!")

greeting\_name(name="Taro")

>> Hello Taro!

関数の呼び出し時に仮引数名=実引数の形で引数を指定する形式を **キーワード引数** と呼ぶ．

def greeting\_name(first, last):

print("Hello " + first + last + "!")

greeting\_name(first="Ichiro", last="Yamada")

>> Hello IchiroYamada!

関数の呼び出し時に仮引数名を省略することができる．  
その際，定義した仮引数の順番に代入される．この形式を **位置引数** と呼ぶ．

def greeting\_name(first, last):

print("Hello " + first + last + "!")

greeting\_name("Jiro", "Yamada")

>> Hello JiroYamada!

仮引数を定義する時に **デフォルト値** を設定することができる.  
ただし，一部の引数にのみデフォルト値を設定する場合は，  
デフォルト値のない引数より後に記述する必要がある．

def greeting\_name(first, last="Yamada"):

print("Hello " + first + last + "!")

greeting\_name("Saburo")

>> Hello SaburoYamada!

**< return文 >**

関数を実行した際に出力される値のことを **戻り値** と呼ぶ．  
関数内に **return文** を記述することで戻り値を指定できる．  
複数の値を指定する場合は,で区切ることでタプルとして返すことが可能．

def split\_complex(complex\_num):

# realメソッドは実部を取得

real\_num = complex\_num.real

# imagメソッドは虚部を取得

imag\_num = complex\_num.imag

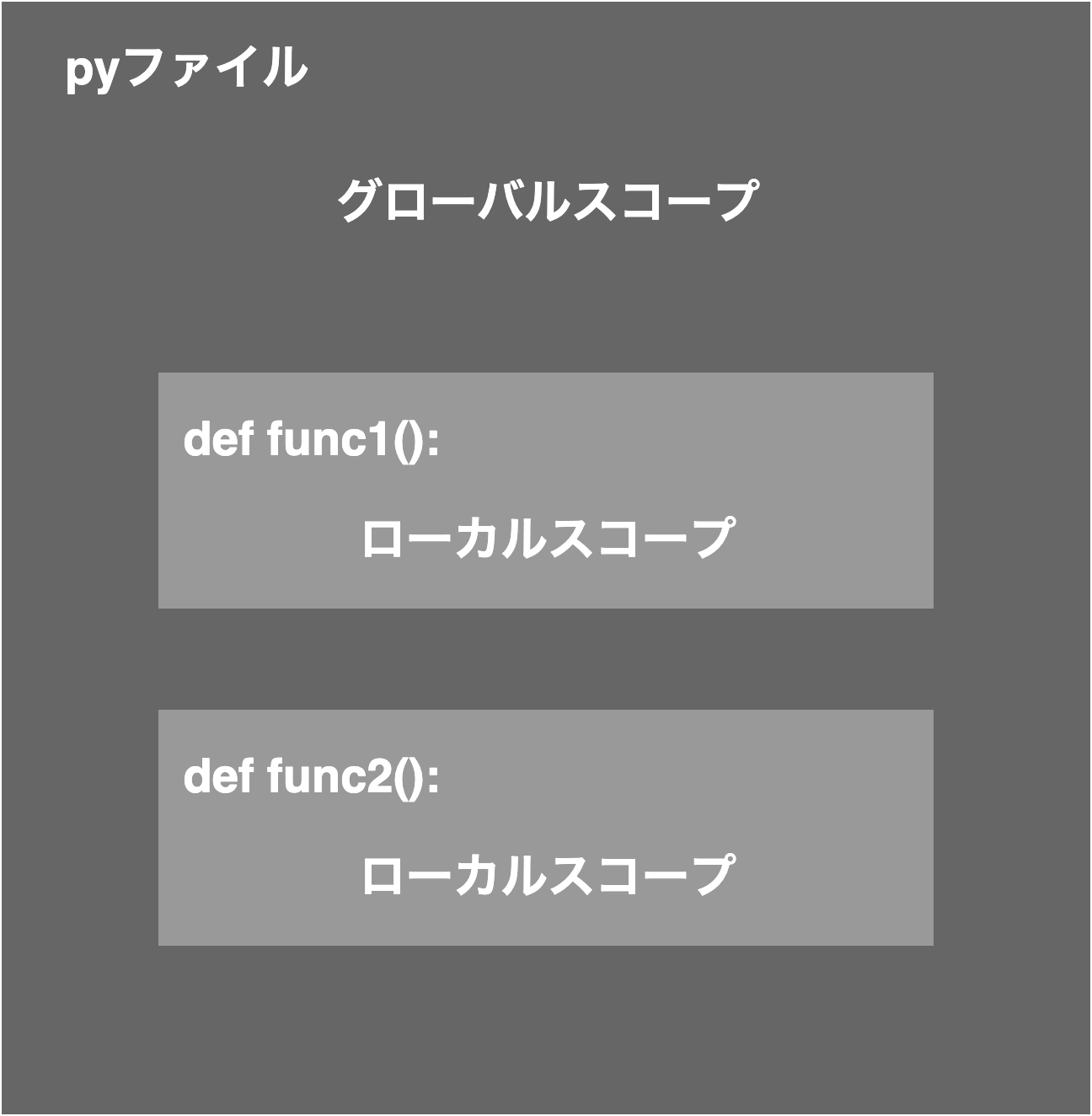
return real\_num, imag\_num

print(split\_complex(2 + 5j))

>> (2.0, 5.0)

**< 名前空間 >**

**名前空間**(スコープ) ... オブジェクトが所属し，使用できる．  
主に **グローバル(モジュール)スコープ** と **ローカルスコープ** の2種類に分かれる．  
グローバルスコープ ... インデントなしで定義されたオブジェクトが所属するスコープ．  
ローカルスコープ ... 関数内で定義されたオブジェクトが所属するスコープ．

[](https://camo.qiitausercontent.com/be286c178ee213e950055cb3b7158a8c34bf900a/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f31666561656135302d393435662d616536612d623533612d3637346432626236626331662e706e67)

そのスコープより上位のスコープに所属するオブジェクトは参照することが可能．  
そのため，グローバルスコープで定義した変数は全ての関数から参照可能である．

hoge = 10

def func():

print(hoge)

func()

>> 10

そのスコープと同階層もしくは下位のスコープに所属するオブジェクトは参照できない．

def func():

hoge = 10

print(hoge)

>> NameError: name 'hoge' is not defined

下位のスコープで変数の値を変更した場合も上位スコープへ影響を与えることはない．

hoge = 10

def func():

hoge = 30

print("function :", hoge)

func()

print("global :", hoge)

>> function : 30

global : 10

**< 関数内関数 >**

def文による関数定義の中にさらに関数を定義することができる．  
関数内関数はその上位関数からしか実行できない．

def outer(hoge, fuga):

def inner(a, b):

return a\*\*2 + b\*\*2

return inner(hoge, fuga)

print(outer(3, 4))

print(inner(3, 4))

>> 25

NameError: name 'inner' is not defined.

**< クロージャ >**

**クロージャ** ... 実行しない状態で記憶する関数オブジェクト．  
引数で渡した情報を保持し， 後から好きなタイミングで実行できる．

def outer(hoge):

def inner():

print(hoge)

return inner

hello = outer("hello")

world = outer("world")

hello()

world()

>> hello

world

クロージャを利用することで，  
異なる初期値を設定した関数を簡単に用意することができる．

def calculate\_area\_circle(pi):

def calculate(radius):

return pi \* radius \* radius

return calculate

calc\_314 = calculate\_area\_circle(3.14)

calc\_3 = calculate\_area\_circle(3)

print(calc\_314(100))

print(calc\_3(100))

>> 31400.0

30000

**< lambda式 >**

def文による関数の定義では関数名を定義する．  
Pythonには名前をつけずに実行できる **無名関数** として *lambda式* が存在する．  
(lambda 仮引数 : 戻り値 )(実引数) の形式で実行できる．

hoge = (lambda a, b: a\*\*2 + b\*\*2)(3, 4)

print(hoge)

>> 25

**< 高階関数 >**

**高階関数** ... 関数を引数に取る関数 ( **map関数** や **filter関数** など )  
map関数 ... リストの各要素に指定した関数を実行し，返す．  
filter関数 ... リストの各要素に指定した関数を実行し，True となるものを返す．

hoge = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

map\_hoge = list(map(lambda x: x\*2, hoge))

print(map\_hoge)

filter\_hoge = list(filter(lambda x: x%2==0, hoge))

print(filter\_hoge)

>> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

[2, 4, 6, 8, 10]

**内包表記**

**< リスト内包表記 >**

次のようなfor文とシーケンスを組み合わせた処理を簡略的に書く記法に **内包表記** がある.

hoge = []

for i in range(5):

hoge.append(i\*2)

print(hoge)

>> [0, 2, 4, 6, 8]

**リスト内包表記** は [] で定義する.  
[式 for [変数名] in [イテラブルオブジェクト]]

print([i\*2 for i in range(5)])

>> [0, 2, 4, 6, 8]

hoge = []

for i in range(10):

if i % 2 == 0:

hoge.append(i)

print(hoge)

>> [0, 2, 4, 6, 8]

リスト内包表記で上記のようなif文による条件分岐を使用する場合は  
[式 for [変数名] in [イテラブルオブジェクト] if [条件式]] のように書く.

print([i for i in range(10) if i % 2 == 0])

>> [0, 2, 4, 6, 8]

hoge = []

for i in range(10):

if i % 2 == 0:

hoge.append(i)

else:

hoge.append('odd')

print(hoge)

>> [0, 'odd', 2, 'odd', 4, 'odd', 6, 'odd', 8, 'odd']

リスト内包表記で上記のようなif-else文を使用する場合は  
[True式 if [条件式] else [False式] for [変数名] in [イテラブルオブジェクト]]  
のように **三項演算子** を用いて書く.

print([i if i % 2 == 0 else 'odd' for i in range(10)])

>> [0, 'odd', 2, 'odd', 4, 'odd', 6, 'odd', 8, 'odd']

**< セット内包表記 >**

**セット内包表記** は {} で定義する.  
{式 for [変数名] in [イテラブルオブジェクト]}

hoge = set()

for i in range(5):

hoge.add(i\*2)

print(hoge)

>> {0, 2, 4, 6, 8}

**< 辞書内包表記 >**

**辞書内包表記** もセット内包表記と同じく {} で定義する.  
{key: value for [変数名] in [イテラブルオブジェクト]}

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

prices = [180, 100, 85]

print({fruit: price for fruit, price in zip(fruits, prices)})

>> {'apple': 180, 'banana': 100, 'lemon': 85}

**< ジェネレータ式 >**

() で内包表記で定義すると **ジェネレータ式** を返す.  
ジェネレータ式は **イテラブルオブジェクト** として使用できる.  
ジェネレータ式ではその都度, **要素を1つずつ生成する** ためメモリを節約できる.  
(処理速度はリスト内包表記の方が速い)

hoge = (i for i in range(5))

for i in hoge:

print(i)

>> 0

1

2

3

4

**イテレータ**

**< イテレータ >**

**イテレータ** ... 要素を1つずつ取り出すことのできるオブジェクト  
リストやタプル, rangeオブジェクト, 文字列, マッピング, 集合などが該当する．  
これらは自身を呼び出す\_\_iter\_\_メソッドと，  
次の要素を呼び出す\_\_next\_\_メソッドを持つ.

hoge = [1, 2, 3].\_\_iter\_\_()

print(hoge.\_\_next\_\_())

print(hoge.\_\_next\_\_())

print(hoge.\_\_next\_\_())

>> 1

2

3

\_\_iter\_\_メソッド, \_\_next\_\_メソッドの代わりに  
**iter関数** , **next関数** を用いることも可能.

hoge = iter([1, 2, 3])

print(next(hoge))

print(next(hoge))

print(next(hoge))

>> 1

2

3

**< ジェネレータ >**

**ジェネレータ** ... イテレータと同じく要素を1つずつ取り出すオブジェクトである.  
ジェネレータはイテレータと違い, その都度要素を生成するため, メモリを節約できる.  
yield式を使うことでジェネレータを生成できる．

def count\_up(num=1):

while True:

yield num

num += 1

c = count\_up()

print(c.\_\_next\_\_())

print(c.\_\_next\_\_())

>> 1

2

**クラス**

**< オブジェクト指向 >**

Pythonは **オブジェクト指向**と呼ばれる考え方に基づいて設計されている．  
オブジェクト指向では， **データ** と **メソッド** を一つのオブジェクトとして定義する．  
例えば, appleという文字列はstr型というオブジェクトとして定義され，  
小文字に変換するlower()，リストに変換するsplit()などのメソッドが紐づいている．

オブジェクト指向と比較される概念に **プロセス指向** がある．  
プロセス指向は，関数を主軸とした設計である．

例として，以下に電話番号を-区切りのリストにする場合のそれぞれの設計の違いを示す．

**オブジェクト指向**

"090-1234-5678".split('-')

* Subject.Verb(Object)の形式を取る
* 対象のオブジェクトに使用できるメソッドが明白である

**プロセス指向**

split("090-1234-5678", '-')

* Verb(Subject, Object)の形式を取る
* 異なるデータ型のオブジェクトにも対応させることが可能

**< class文 >**

**class文** を使用することで新たにオブジェクトの **設計図** を定義することが可能．  
class クラス名: で定義し， 変数 = クラス名() でオブジェクトの実体を生成する．  
このオブジェクトの **実体** を **インスタンス** と呼ぶ．

class Student:

pass

hoge = Student()

**< クラス変数 >**

**クラス変数** ... クラス自身にデータを保持するための変数．  
クラス変数は全てのインスタンスに共有される．

class Student:

univ = "Musashino"

print(Student.univ)

hoge = Student()

print(hoge.univ)

>> Musashino

Musashino

**< アトリビュート >**

**アトリビュート** ... インスタンス自身にデータを保持するための変数．  
別名を **インスタンス変数** という．  
インスタンス.アトリビュート名 = 値 でアトリビュートを定義できる．

class Student:

pass

taro = Student()

taro.id = 1

taro.name = "taro"

print(taro.id, taro.name)

>> 1 taro

**< コンストラクタ >**

**コンストラクタ** ... インスタンスを生成した時に一度だけ実行されるメソッド．  
インスタンス生成時にアトリビュートの設定をする際などに使用する．  
コンストラクタは def \_\_init\_\_(self, 仮引数): で定義する．  
self はインスタンス自身を表し，実引数では無視する．

class Student:

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

self.id = id

self.name = name

taro = Student(1, "taro")

print(taro.id, taro.name)

>> 1 taro

**< メソッド >**

**メソッド** ... クラス内で定義される関数であり，そのクラスのインスタンスのみに実行可能．  
コンストラクタ同様にインスタンス自身を操作するために，第一引数にはselfを指定する．

class Student:

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

self.id = id

self.name = name

def print\_age(self, age):

print(self.name + "の年齢: " + str(age))

taro = Student(1, "taro")

taro.print\_age(20)

>> taroの年齢: 20

**< デストラクタ >**

**デストラクタ** ... インスタンスを削除する際に呼び出されるメソッド．  
デストラクタは def \_\_del\_\_(self, 仮引数): で定義する．  
インスタンスの削除には **del文** を用いる．

class Student:

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

self.name = name

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

print(self.name + "のインスタンスを削除")

taro = Student("taro")

del taro

>> taroのインスタンスを削除

**< 継承 >**

**継承** ... 既に定義したクラスのアトリビュートやメソッドを引き継ぎ定義すること．  
class 子クラス名(親クラス名): で定義する．

class Parent:

first, last = "Taro", "Yamada"

def print\_name(self):

print(self.first + self.last)

class Child(Parent):

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

self.first = first

hanako = Child("Hanako")

hanako.print\_name()

>> HanakoYamada

**< 多態性 >**

**多態性(ポリモーフィズム)** ... 継承した上で異なる振る舞いを持つこと．  
一部のメソッドを上書きする **オーバーライド** により，異なる振る舞いを実現する.  
[](https://camo.qiitausercontent.com/78379a395345806f34038a4c0971a84b999abbda/68747470733a2f2f71696974612d696d6167652d73746f72652e73332e61702d6e6f727468656173742d312e616d617a6f6e6177732e636f6d2f302f313333343038322f31626238626535342d313032612d333632392d303335612d3465323365623239663637362e706e67)

class Weapon():

def attack(self):

print("攻撃する")

def drop(self):

print("落とす")

class Gun(Weapon):

def attack(self):

print("撃つ")

class Knife(Weapon):

def attack(self):

print("刺す")

class Hammer(Weapon):

def attack(self):

print("叩く")

g = Gun()

g.attack()

g.drop()

k = Knife()

k.attack()

k.drop()

h = Hammer()

h.attack()

h.drop()

>> 撃つ

落とす

刺す

落とす

叩く

落とす

**参考**

[Python ドキュメント : https://docs.python.org/ja/3/](https://docs.python.org/ja/3/)  
[pep8-ja ドキュメント : https://pep8-ja.readthedocs.io/ja/latest/](https://pep8-ja.readthedocs.io/ja/latest/)