# 1 もくじ

- <u>Pythonの実行</u>
- Hello, world!
  - o print関数について
- 変数
  - 。 変数を作る
- 型
  - 。 今回学ぶ型の種類
- 数值型
  - 。 概要
  - 。 <u>int型と四則演算</u>
  - 。 int型の上限
  - 。 <u>小数点の扱</u>い
  - 。 複素数
  - <u>累乗, ルート</u>
  - 。 代入演算
  - 。 比較演算
  - 。 intとbit演算
- ブール値
  - 。 概要
  - 。 ブール値とブール演算
- シーケンス型
  - 。 概要
  - 。 list(リスト)型の説明
- <u>マッピング型</u>
  - 。 概要
  - 。 辞書(dict)の作り方/呼び出し方
  - 。 辞書の順序
- 変数確保の仕組み

# 2 Pythonの実行

#### Pythonの実行方法は

- 1. ファイルを読み込み, それを逐次実行していく方法
- 2. 対話形式で1文ずつ実行していく方法

の二つがあります.

```
1では .py ファイルを作成し, $ python file_name.py のように読ませます. 2では $ python と入力すると, 対話モードに入り, 逐次実行できるようになります.
```

# 3 Hello, world!

まずは最初に "hello, world!" を表示していきましょう. C言語なら以下のとおりです.

```
#include <stdio.h>

void main(void) {
    printf("hello, world!\n");

    return;
}
```

次にPythonで書いていきます. (作成環境上, pythonだけ結果を出力していきます.)

```
print("hello, world!")
hello, world!
```

このようにPythonでは main 関数を利用する必要がありません. また,標準出力関数 print は末尾に改行が入るので \n を入れる必要がありません.

ただ, このように平らな土地にプログラムを書いていくと, 個人的にスコープが汚れていくのが許せないので, 以下のような書き方をしていきたいと思います.

```
def main():
    print("hello, world!")
```

```
if __name__ == '__main__':
    main()

hello, world!
```

以降, このようにmain関数を作ってそこで主にプログラムを書く形式を取っていきます. (なぜこのような書き方をするかは後で詳しく説明します.)

## print関数について

print関数を今後良く使っていくと思うので、使い方を少しだけ補足します.



#### 改行について

「末尾に改行が入る」と言いましたが、オプションで変えられます.

```
print("hoge", end="")
print("fuga", end="$$")
print("piyo", end="卍")
```

hogefuga\$\$piyo卍

引数 end には文章の末尾に置く文字を選択できます. 末尾に改行が入っているのは引数 end にデフォルトで \n が入っているためです.

#### 表示できるもの

3 hoge 4.6 [3, 4, 5]

以下のように , で区切って渡せば, それらが半角スペース1つ文間隔で表示されます. 引数の型は文字列だろうと数字だろうと配列だろうと, いい感じに表示してくれるので安心して渡していきます.

```
print(3, "hoge", 3.4 + 1.2, [3, 4, 5])
```

## 4 変数

## 変数を作る

- 作成するプログラム
- 整数値 10 を入れる変数を作成する.
- 小数値 8.2 を入れる変数を作成する.
- 文字列 "hoge" を入れる変数を作成する.
- 最後にそれらを表示する.

### C言語での実装

```
int val1 = 10;
double val2 = 8.2;
char str[] = "hoge";

printf("%d, %lf, %s\n", val1, val2, str);
```

### pythonでの実装

```
val1 = 10
val2 = 8.2
string = "hoge"

print(val1, val2, string)
```

```
10 8.2 hoge
```

と書ける.; は必要ないです.

Pythonの変数宣言は型を指定する必要がない. また,以下のように変数への値の再代入において,変数と入れる値が同じ型かどうかを気にする必要がない.

```
a = 10;    print(a)
a = 3.2;    print(a)
a = "hoge";    print(a)

10
3.2
hoge
```

ちなみに、; で区切ることで、1行に複数の命令を書ける.

### 5型

### 今回学ぶ型の種類

Pythonではあまり型を気にする必要がないのですが,一応かる一く触れておきます.

- 数值型
  - o int
  - float
  - complex
- ブール型
- シーケンス型
  - list
  - tuple
  - 。 文字列
- マッピング型
  - dict

他にもいろいろありますが、以上の型は最低限覚えておきましょう. それぞれについてC言語との違いに触れつつ見ていきます.

## 6数值型

### 概要

ここでは数値型の説明と、それらの演算について説明します.数値型は以下の3つです.

- int
- float
- complex

### √int型と四則演算

#### 作るもの

- a = 10, b = 3とする.
- a+bを表示
- *a* − *b* を表示
- a×bを表示
- a÷bの商, 剰余を表示
- *a* ÷ *b* を小数で表示

#### C言語なら

```
int a = 10, b = 3;

printf("和 = %d\n", a + b);
printf("差 = %d\n", a - b);
printf("積 = %d\n", a * b);
printf("(商, 剰余) = (%d, %d)\n", a / b, a % b);
printf("小数 = %lf\n", (double)a / b);
```

### Pythonなら

```
a = 10; b = 3
```

```
print( "和 =", a + b )

print( "差 =", a - b )

print( "積 =", a * b )

print( "(商, 剰余) = ({}, {})".format(a//b, a % b))

print( "小数 =", a / b)
```

/ の演算が少し異なります. C言語では, int / int 3^5 = 243 3^(1/3) = 1.44225は int で出るため, 商が出ます. しかし, Pythonでは / は余りを出さずに割る演算なので小数点まででます.

もしPythonで賞を出したいときは // を使います.

ちなみに型は以下のように見れる.

```
a = 10
print( a.__class__.__name__ )
```

もちろん int 型

int

### int型の上限

- 作るもの
- 99999999999999999999999 を変数にいれ,表示する.
- C言語なら

intだと余裕でオーバーフローしそうなので long int を利用してみる.

```
long int a = 999999999999999999999;
printf("%ld\n", a);
```

#### 出力結果

7766279631452241919

結局オーバーフローします.



### Pythonなら

```
a = 9999999999999999
print( a )
```

99999999999999999

ふつーに表現できる(オーバーフローしない).
Pythonではint型なら表現の限界がない. (一応, メモリ確保できうる最大限という制約はあるけど)

#### すごい影

### 小数点の扱い



#### 作るもの

小数点の四則演算

• 1.9999999999999を変数に入れ,表示する.



#### C言語



#### Python

Pythonの小数点数の型は float のみで, C言語でいう double であり, デフォルトで倍精度を保証している. もちろん, こちらは表現の限界がある. そのため, 先ほどのように表現限界が来て 2.0 となる.

ちなみに今回も型を見てみると,

```
a = 12.3
print( a.__class__.__name__ )
```

float

### 複素数



やること

複素数の四則演算



#### C言語だと

かなりめんどくさいのでパス.



### Pythonだと

```
a = 3.2 + 4.1j; b = 7.5 + 3.9j

print( "a + b ->", a + b )

print( "a - b ->", a - b )

print( "a * b ->", a * b )

print( "a / b ->", a / b )
```

j を数値の後ろに足すことで, 虚数の表現が可能. (数値が先頭にくることで変数ではないと解釈できる.)

この型を見てみると

```
a = 10 + 3j
print( a.__class__.__name__ )
```

complex

complex 型でした.

# ▲ 累乗, ルート



### 作るもの

- 1. aに任意の整数を入れる
- 2. aのN1乗を出力
- 3. aのN2乗根を出力



#### C言語では

```
#include<math.h>
#include<stdio.h>

void main(){
    int a = 3, N1 = 5, N2 = 3;
    printf( "%d^%d = %g\n", a, N1, pow((double)a, N1) );
    printf( "%d^(1/%d) = %g\n", a, N2, pow((double)a, (1.0/N2)) );
}
```

```
3^5 = 243
3^(1/3) = 1.44225
```



### Pythonでは

```
print( 3 ** 5)
print( 3 ** (1/3))
```

1.4422495703074083

Pythonでは煩わしい関数を使わずに \*\* でべき乗が表せる.

### 代入演算

代入演算子は、特に制作物も思いつかないので、紹介だけしておきます. 代入演算子は a = a (演算子) n のような演算をより簡潔に書くための演算子です. a = a (演算子) n は a (演算子) n のように書き直すことができます.

```
a = 10

a += 12; print(a) # ( a += 12) <-> ( a = a + 12 )

a %= 3; print(a) # ( a %= 3 ) <-> ( a = a % 3 )

a /= 3.4; print(a) # ( a /= 3.4) <-> ( a = a / 3.4 )

a *= 2; print(a) # ( a *= 2) <-> ( a = a * 2 )
```

22

1

0.29411764705882354

0.5882352941176471

### 比較演算

各数値同士を比較し, その結果をbool値(True か False の二値)で返す演算です.

演算例	True なら
a == b	a が b と等しい
a != b	a が b と等しくない
a < b	a が b より小さい
a > b	a が b より大きい
a <= b	aがb以下

演算例	True なら
a >= b	a が b 以上
a 💠 b	a が b と等しいくない
a is b	a が b と等しい
a is not b	a が b と等しくない
a in b	a が b の中に含まれる
a not in b	a が b の中に含まれない

```
a = 10; b = 23; print( a > b , a < b )
a = 10; b = 10; print( a >= b , a <= b )

a = 10; b = 20; print( a == b, a != b )</pre>
```

False True

True True

False True

### intとbit演算

たぶんmplをいじる上では必要ないので省略します.

# 7ブール値

### 概要

ブール値(True と False の二値)を取る型である ブール値 について説明します.

## ブール値とブール演算

- 作るもの
- 整数変数a, b, cを作成

- a > b , b > c , c > a を表示
- a > b , b > c の and , or 演算を行い, 表示する
- c > a の not 演算を行い, 表示する

### C言語

```
int a = 3, b = 4, c = 2;

printf( "a > b = %d, b > c = %d, c > a = %d\n", a > b, b > c, c > a );

printf( "a > b && b > c = %d\n", a > b && b > c );

printf( "a > b || b > c = %d\n", a > b || b > c );

printf( "!(c > a) = %d\n", !(c > a) );
```

```
a > b = 0, b > c = 1, c > a = 0

a > b && b > c = 0

a > b &|| b > c = 1

!(c > a) = 1
```

C言語にはブール型というものがなく,比較演算の結果が偽なら 0,真なら 0以外 (主に1)となっている.



### Python

```
a = 3; b = 4; c = 2

print( "a > b =", a > b, "b > c =", b > c, "c > a =", c > a);
print( "a > b and b > c =", a > b and b > c )

print( "a > b or b > c =", a > b or b > c )

print( "not (c > a) =", not (c > a) )
```

```
a > b = False b > c = True c > a = False
a > b and b > c = False
```

```
a > b or b > c = True
not (c > a) = True
```

各演算の対応は以下のように成っている.

演算内容	C言語	Python
and演算	&&	and
or演算	11	or
否定	!	not

# 8 シーケンス型

### 概要

とりあえず,以下のものだけわかっておけば問題ないと思います.

型の名 前	内容	C言語で言うと
list	いくつかの要素を内包したミュータブルなオブジェ クト	型はなんでもOKなmalloc不要の配 列
tuple	listのイミュータブル版	
文字列		

また, この回に関しては演習が少し作成しづらいので, 文章だけの説明になります. for 文のところで嫌という程使うので安心してください.

### list(リスト)型の説明

C言語の配列をめっちゃくちゃ柔軟にしたみたいなやつ.

各要素のアクセスはC言語と同様.

(またあとで説明しますが、アクセス方法はシーケンス型なら全て共通している.)

listの作成には [] を利用し, 要素を , で区切って表現していく.

```
a = ["hoge", "fuga", "piyo"]
print(a, a[0], a[1], a[2])
```

```
['hoge', 'fuga', 'piyo'] hoge fuga piyo
```

違う型の混在もできるし、ネストもできる.

```
a = ["hoge", [1, 2], [[3, 4], [5, 6]]]
print(a[0])
print(a[1], a[1][0], a[1][1])
print(a[2])
print(a[2][0], a[2][0][0], a[2][0][1])
print(a[2][1], a[2][1][0], a[2][1][1])
```

```
hoge
[1, 2] 1 2
[[3, 4], [5, 6]]
[3, 4] 3 4
[5, 6] 5 6
```

#### もちろん各要素の変更も可能

```
a = ["hoge", [1, 2], [[3, 4], [5, 6]]]
a[2] = "fuga"
print(a)

a[1][1] = "piyo"
print(a)
```

```
['hoge', [1, 2], 'fuga']
['hoge', [1, 'piyo'], 'fuga']
```



tuple(タプル)型の説明

誤解を恐れず言うのであれば、リストの各要素の書き換え/追加/削除等の変更操作が不可能になったもの. () でくくって表現する.

```
a = ("hoge", [1, 2], [[3, 4], [5, 6]])
print(a)

('hoge', [1, 2], [[3, 4], [5, 6]])
```

置き換えができないとはいったが,要素自体の変更ができないだけで,その要素自体がミュータブル(書き換え可能)なら,要素の要素が変わるのは問題ない.

```
a = ("hoge", [1, 2], [[3, 4], [5, 6]])
# a[0] = 2 や a[1] = 10 などは不可だけど…
a[1][0] = 99

print(a)
```

```
('hoge', [99, 2], [[3, 4], [5, 6]])
```

#### 文字列

文字列は文字単位の集まりと考えられ、シーケンス型の1つである.

"" (ダブルクオーテーション)か '' (シングルクオーテーション)で囲うことで, 文字列を表現できる.

```
print( "OK\n牧場" )
```

0K

牧場

二つに機能的な差はないので, 使い分けとしては文字列中に \*\* や \*\* を使うかどうかである. 片方が登場する場合はもう片方を文字列の囲みに利用すれば良い.

```
print( " ' はシングルクオーテーション" )
print( ' " はダブルクオーテーション' )
```

- ' はシングルクオーテーション
- " はダブルクオーテーション

また, """ ~~ """ のようにダブルクオーテーション3つでヒアドキュメントという機能が使える. これは囲った文字列をそのまま取り込むもので, 改行も一度に取り込めるので, 複数の改行がある時に便利(かもしれない).

```
      str = """私は今

      <td rowspan="2" with the string of the
```

私は今 呼吸をして いるかもしれない. やっぱりしてない かもしれない.



#### シーケンス型のアンパック

シーケンス型のように連続した要素を持つものを,分解していくつかの変数へ代入する操作を アンパック という.

```
a, b, c = [30, 50, 70]

print(a, b, c)

a, b, c = ("hoge", 30, 3.14)
```

```
print(a, b, c)

a, b, c = "get"
print(a, b, c)

30 50 70
```

```
30 50 70
hoge 30 3.14
g e t
```

また,シーケンスの要素数と,左辺の代入対象の変数の数は一致しないといけない.



### シーケンスと演算子

```
# 演算子 * => 複製
a = [1, 2, 3] * 3
b = (1, 2, 3) * 2
c = "abc" * 4
print(a)
print(b)
print(c)
print()
# リスト/タプルと演算子 + => 結合
a = [1, 2, 3] + [4, 5, 6]
b = (1, 2) + (3, 4)
c = "abc" + "def"
print(a)
print(b)
print(c)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
(1, 2, 3, 1, 2, 3)
abcabcabca

[1, 2, 3, 4, 5, 6]
(1, 2, 3, 4)
abcdef
```

#### シーケンス型における要素へのアクセス方法(1つの要素)

C言語と同じような感じで取ってこれる. ちなみに一番後ろの要素は -1 でとれ, 以降 -2 , -3 でとってこれる.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print( a[5], a[2] )
print( a[-1], a[-3] )
```

6 3

9 7



#### シーケンス型における要素へのアクセス方法(スライスの利用)

[start:end:step] というもの(スライス)を使い,0個以上の要素をかなり柔軟に取ってくることができる.

- start => 取ってきたい最初の位置 (デフォルト 0)
- end => 取ってきたい最後の位置+1 (デフォルト シーケンスの要素数)
- step => 何個おきに取ってくるか ( デフォルト 1 )

基本的な使い方は以下の通り.

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

print(a[3:6]) # 1つおき

print(a[1:6:2]) # 2つおき
```

```
[4, 5, 6]
[2, 4, 6]
```

また、start は省略するとデフォルト値 0 を取り、end は省略するとデフォルト値 vartheta va

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

print( a[:] ) #全て

print( a[5::2])

print( a[::2])
```

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

step には負の数を持ってくることができるため, 逆順も可能.

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

print( a[7:2:-1] )
```

```
[8, 7, 6, 5, 4]
```

また, 単一の要素を選択するときと同様に, start / end に負の数を選択することができ, 後ろの要素から選択することも可能.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

print( a[2:-2:3] )
```

```
[3, 6, 9]
```

ちなみに範囲外をスライスの範囲にしていすると

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

print( a[12:40:3])
```

空のシーケンスが得られる. 値の参照のときとは違い, スライスの取得の際はエラーが出ないので覚えておくと役に立つ.



#### シーケンスのネスト

シーケンス内にシーケンスを入れることができ、その値を参照する場合 [] を連ねて書く. 外側の要素から内側の要素を選んでいく感じで [] を連ねていく.

```
a = ["0", "1", ["2-0", "2-1", ["2-2-0", "2-2-1"]], "4", ["5-0", "5-1"]]
print(a[0])
print(a[2], "@", a[2][0], "@", a[2][1])
print(a[2][2], "@", a[2][2][0], "@", a[2][2][1])
```

```
0

['2-0', '2-1', ['2-2-0', '2-2-1']] @ 2-0 @ 2-1

['2-2-0', '2-2-1'] @ 2-2-0 @ 2-2-1
```

# 9 マッピング型

### 概要

マッピング型はハッシュ可能なオブジェクトをキーとして,任意のオブジェクトを特定できるもの. いまのところマッピング型は dict しかないので,マッピング型= dict という認識で良いと思います.

## 辞書(dict)の作り方/呼び出し方

{key1:value1, key2:value2 …} という感じで作る.

key を入れることで value が呼び出されるようになっており、key はハッシュ可能でなければならないので,他の key と重複してはいけない. (value は重複可)

```
dictionary = {10: 30, "hoge":4.4, 3.4:"fuga"}

print(dictionary[10])
print(dictionary["hoge"])
print(dictionary[3.4])
```

30

4.4

fuga

### 辞書の順序

Python3.7以降では,辞書の要素の順番を保持するようになっている(らしい) (以前までは順番を保持しておらず,順序が実行のたびに変わっていた.)

# 10 変数確保の仕組み

余談になりますが、C言語での変数のメモリ確保の挙動とPythonでの挙動を見ていきたい思います. ひょっとしたらむしろこんがらがるだけなので、飛ばしたければ飛ばしても大丈夫です.

#### 作成中…