#Days 06：Python 入門

- #Days 06：Python 入門
  - Python プログラミング環境の選択肢 3 つ
  - 名前，アドレス，値
    - object のアドレス　idobject
    - object の型　typeobject
  - コメントアウト
  - 数値と数字の変換
    - 数値 ⇒数 字
    - 数字 ⇒数 値
      - 整数値 ⇒ 2 進数表記，8 進数表記，16 進数表記の数 字 列
      - 進数表記，8 進数表記，16 進数表記の数字 列 ⇒ 10 進数表記の数 値
  - 文字コード
    - 文字⇒ユニコード表⇒整数　ord 文字
    - 整数⇒ユニコード表⇒文字 chr 整数
    - 16 進数表記の整数　⇒　文字
    - 値⇒字列
  - エラー Error：構文エラー Sintax Error と例外 Exception
    - 構文エラー
    - 例外
    - 例外処理 1 ：try-except-else
    - 例外処理 2 ：try-except-except-else
    - 例外処理 3 :as の利用
    - 例外処理 4 ：同じ例外処理
    - 例外処理 5 ：一般例外処理
    - 例外処理 6 ：try-except-else-finally
  - 演算子
    - 算術演算子
    - 比較演算子値と値の関係が正ければ True，間違っていれば False
      - >, >=, ==, <=, <, !=
      - is , is not（後日，「リスト」を学習する際にまた説明します）
    - 論理演算子
      - 前提
      - and は，「かつ」
      - or は「または」
      - not はブール反転
    - ビット演算子

---

0. Python プログラミング環境の選択肢 3 つ

1. ソースコード `hogehoge.py` を書く -> 書き終えたら端末から `python hogehoge.py` ，バッチ実行
2. ソースコードは作る気がない -> いきなり端末で `python`(Python Terminal)，行単位実行

3. 1 と 2 のハイブリッド．つまり，ソースコードを書かないで始める-> `ipython`(Jupyter, GoogleColab), セル単位実行-> コードを保存

それぞれやってみます．Windows でも Linux でも Mac でも同じです．（演習 I でやったみたいだけど）

---

1. 名前，アドレス，値

`name = value`
という構文は，「値 value に名前 name をつけて，address に記憶する」の意味．

1.1. object のアドレス　`id(object)`

```
>>> print(id(x)) # 素の x にはアドレスが無い．
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined

>>> print(id(3)) # 素の数値には適当なアドレスがある．
140734273462736

>>> x = 3
>>> y = 4

>>> print(id(x)) # 名前として記憶すると，数値のアドレスになる．
140734273462736

>>> print(id(y)) # 32bit 進んだアドレスになる．先ほどの数値は 32 ビット幅だということ
140734273462768

>>> z = x

>>> print(id(x)) # 別の数値の名前にならない限り，同じアドレス
140734273462736

>>> print(id(z)) # 数値 3 に対する名前は x と z だということ
140734273462736

>>> x = 5
>>> print(id(x)) # (y,4) から 32 ビット進んだアドレスに付け変わる．
140734273462800
```

1.2. object の型　`type(object)`

数字リテラルは数値そのもの，クォーテーションで囲んだ数字は文字列．

```
>>> type(3)
<class 'int'>

>>> type(3.14)
<class 'float'>

>>> type("3")
<class 'str'>

>>> type("3.14")
<class 'str'>

>>> type(True)
<class 'bool'>
```

int, float など「型名」の型は，'type'

```
>>> type(int)
<class 'type'>
```

リテラル x は型がないが，値に対応づけると，名前になり，型はその値と同じになる．

```
>>> type(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 1
>>> type(x)
<class 'int'>
```

---

2. コメントアウト

ソースコード中に無視してほしい構文を書いたところを「コメント」といいます．ソースコードの一部をコメント扱いにするとき，「コメント扱いにする」という操作を「コメントアウト」といいます．

python 言語では，2 種類のコメントアウト方法があります．

もとが

```
x + y
y + z
z + x
```

- # は，この記号以降，改行するまでコメント扱い

```
x + y
y # + z
z + x
```

は

```
x + y
y
z + x
```

と同じです.

- ''' (3 連続 single quatation) と """ (3 連続 double quatation) は，行単位で囲まれる部分をコメント扱い

```
x + y
'''
y + z
'''
z + x
```

は

```
x + y
z + x
```

と同じです.

---

3. 数値と数字の変換

3.1. 数 値 ⇒数字

```
>>> str(3)
'3'
>>> str(3.0)
'3.0'
```

3.2. 数 字 ⇒数値

```
>>> int("3")
3
>>> float('3')
3.0
```

```
>>> str(97)
'97'
>>> bin(97)
'0b1100001'
```

```
>>> oct(97)
'0o141'
>>> hex(97)
'0x61'
```

整数 値⇒ **2** 進数表記，**8** 進数表記，**16** 進数表記の数 字列

- 0b(ゼロ・ビー)：b of binary(2 進数)
- 0o(ゼロ・オー)：o of octet（8 進数）
- 0x(ゼロ・エックス)：x of hextet(16 進数）

2 進数表記，**8** 進数表記，**16** 進数表記の数字 列 ⇒ **10** 進数表記の数値　`int(str, base)` 関数は，str を基数 base の数値と捉えて，10 進数表記に変換

```
>>> int("11000001",2) # 11000001 を 2 進数の数値と捉えて，10 進数表記に変換
193
>>> int("11000001",3) # 11000001 を 3 進数の数値と捉えて，10 進数表記に変換
2917
>>> int("11000001",8) # 11000001 を 8 進数の数値と捉えて，10 進数表記に変換
2359297
>>> int("11000001",10) # 11000001 を 10 進数の数値と捉えて，10 進数表記に変換
11000001
>>> int("11000001",16) # 11000001 を 16 進数の数値と捉えて，10 進数表記に変換
285212673
```

基数が無くても，0b をつければ binary として，0o をつければ octet として，0x をつければ hextet として扱う

```
>>> int('97')
97
>>> int('0b1100001')
97
>>> int('0o141')
97
>>> int('0x61')
97
```

```
>>> 0b1100001  # ソースコードでは print(0b1100001)
97
>>> 0o141
97
>>> 0x61
97
```

---

4. 文字コード

4.1. 文字⇒ユニコード表⇒整数　ord(文字)

```
>>> ord("a")
97

>>> ord("あ")
12354
```

この整数は 16 進数で表すこともある

```
>>> hex(ord("あ"))
0x3042

>>> ascii('あ')
"'\\u3042'"
```

4.2. 整数⇒ユニコード表⇒文字 chr(整数)

```
>>> chr(97)
'a'

>>> chr(12354)
'あ'
```

4.3. 16 進数表記の整数　⇒　文字

```
>>> '\u3042'
'あ'
```

4.4. 値⇒字列

値⇒字列　は s.format(spec) というメソッドを使うともっと柔軟に扱えます．のちほど，別の機会に説明します．

---

5. エラー (Error)：構文エラー (Sintax Error) と例外 (Exception)

ソースコードは，行ごとに上から順に処理されていきますが，途中で止まってしまうことがあります．それをエラーといいます．

エラーには 2 種類あります．

- 構文エラー（所謂，「エラー」, Syntax Error）：構文の誤り（文法上の誤り）
- 構文エラー以外（所謂，「例外」，Exception）

5.1. 構文エラー

「Syntax Error」と表示されるエラーです．致命的なエラーなので必ず修正しなければなりません．
例えば，

```
>>> s = "Hello World.
```

これを実行すると

```
  s = "Hello World.
                   ^
SyntaxError: EOL while scanning string literal
```

と表示されて停止します．「文字列リテラルの終端を探していたら行末になってしまった．」
これ以外にも SyntaxError の具体的なメッセージはいろいろあります．

5.2. 例外

「SyntaxError」以外の表示で停止することがあります．それをまとめて「例外」と言います．
例えば，

```
# foo.py

a = 10
b = 0
# b = 'hello'

c = a / b
print(c)
print('divise done')

d = a * b
print(d)
print('multiply done')
```

これを実行すると，ゼロで割っているよ．というエラーで停止し，それあとの `print(c)` 以降が処理されません．

```
> python days06/foo.py
Traceback (most recent call last):
  File "days06/foo.py", line 7, in <module>
    c = a / b
ZeroDivisionError: division by zero
```

他にも例えば

```
# bar.py

y = 2
    x = 1
```

これを実行すると，インデントが揃っていないよ，というエラー

```
> python days06/bar.py
  File "days06/bar.py", line 4
    x = 1
    ^
IndentationError: unexpected indent
```

例外は致命的なエラーではなく，警告に過ぎません．修正の必要はないですが，停止させないようにするには，「例外処理」で包む必要があります．

5.3. 例外処理１：try-except-else

例えば，最初の ZeroDivisionError の例では，

```
# foo-rev.py

a = 10
b = 0
# b = 'hello'

try:
  c = a / b
except(ZeroDivisionError):
  print("b cannot be zero")
else: # 例外エラー以外の処理
  print(c)

print('divise done')

d = a * b
print(d)
print('multiply done)
```

と例外処理で包んでやると停止せずに続けてくれます．
try のブロックに print(c) を加えてはいけません．try のブロックは例外エラーが起きる該当行だけを書いて例外エラーを捕まえるだけです．

```
> python days06/foo-rev.py
b cannot be zero
divise done
0
multiply done
```

5.4. 例外処理２：try-except-except-else

b = 'hello' だったらどうでしょう．
この場合も，実行したとき，例外エラー TypeError（型エラー）が表示され，停止します．

```
> python days06/foo.py
Traceback (most recent call last):
  File "days06/foo.py", line 7, in <module>
    c = a / b
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

これも例外処理しましょうか.

```
# foo-rev2.py

a = 10
# b = 0
b = 'hello'

try:
  c = a / b
except(ZeroDivisionError):
  print("b cannot be zero")
except(TypeError):
  print("unsupported operand for /")
else:
  print(c)

print('devise done')

d = a * b
print(d)
print('multiply done')
```

```
> python days06/foo-rev2.py
unsupported operand for /
devise done
hellohellohellohellohellohellohellohellohellohello
multiply done
```

ちなみに，`d = 10*'hello'` は 10 回 `hello` を繋げたものになります.

5.5. 例外処理 3 :as の利用

こんな書き方もできます.

```
# foo-rev3.py

a = 10
# b = 0
b = 'hello'
```

```
try:
  c = a / b
except(ZeroDivisionError) as ErrroMessage:
  print(ErrorMessage)
except(TypeError) as ErrorMessage:
  print(ErrorMessage)
else:
  print(c)

print('devise done')

d = a * b
print(d)
print('multiply done')
```

as 変数をつけると，例外エラーのメッセージが変数に記憶されます．なので，その結果，

```
> python days06/foo-rev3.py
unsupported operand type(s) for /: 'int' and 'str'
devise done
hellohellohellohellohellohellohellohellohellohello
multiply done
```

と表示されます．

5.6. 例外処理４：同じ例外処理

例外処理が同じなので，まとめることもできます．

```
# foo-rev4.py

a = 10
# b = 0
b = 'hello'

try:
  c = a / b
except(ZeroDivisionError,TypeError) as ErrroMessage:
  print(ErrorMessage)
else:
  print(c)

print('devise done')

d = a * b
print(d)
print('multiply done')
```

```
> python days06/foo-rev4.py
unsupported operand type(s) for /: 'int' and 'str'
devise done
hellohellohellohellohellohellohellohellohellohello
multiply done
```

5.7. 例外処理５：一般例外処理

他の例外エラーもあるかもしれません．そのときは例外エラーの名前ではなく，その代表として Exception を使います．

```
# foo-rev5.py

a = 10
# b = 0
b = 'hello'

try:
  c = a / b
except Exception as ErrroMessage:
  print(ErrorMessage)
else:
  print(c)

print('devise done')

d = a * b
print(d)
print('multiply done')
```

```
> python days06/foo-rev5.py
unsupported operand type(s) for /: 'int' and 'str'
devise done
hellohellohellohellohellohellohellohellohellohello
multiply done
```

5.8. 例外処理６：try-except-else-finally

print('devise done') はブロックとしては，try-except-else のブロックにまとめるべきかもしれません．

```
# foo-rev6.py

a = 10
# b = 0
b = 'hello'
```

```
try:
  c = a / b
except Exception as ErrorMessage:
  print(ErrorMessage)
else:
  print(c)
finally: # 例外でも正常でも最後におこなわれる
  print('devise done')

d = a * b
print(d)
print('multiply done')
```

```
> python days06/foo-rev6.py
unsupported operand type(s) for /: 'int' and 'str'
devise done
hellohellohellohellohellohellohellohellohellohellohello
multiply done
```

---

6. 演算子

6.1. 算術演算子

数値どうしの演算に用いられます．

- 四則演算+,-,*,/は基本通り．ただし除算について，
  - / 実数割り算
  - // 整数割り算，商
  - % 整数割り算したときの余り（剰余，合同数，モジュロ，mod）

```
>>> 3.14 / 2
1.57
>>> 3.14 // 2
1.0
>>> 3.14 % 2
1.1400000000000001
```

- ** べき乗，累乗

```
>>> 3.14 ** 2
9.8596
```

- @ 行列乗算: 行列は未だ出てきていませんが，行列 M, ベクトル x, y に対して，

```
>>>  y = M @ x
```

のように計算されます.

優先順位は，カッコ＞累乗＞乗除＞加減　の順ですよね．次の式の答えはいくつでしょう？？

```
>>> 0 - (1 + 2) + 3 ** 4 * 5 / 6
```

答えは，**64.5** です．（正解しましたか？？）

また，プログラミング特有の優先順位も追加されます．それは「同じ優先順位なら左から」．書いてある順番にしたがって計算するという常識的なことに思えます．しかし数学的には，さきほどの例の $0-(1+2)+3^4*5/6$ は，表記の順番を取り替えた $3^4/6*5+0-(2+1)$ と同じになります．だけれども，計算機では演算が一つ終わるごとに「丸め」「情報落ち」が発生するので，行全体の答えが同じにならないことがあるということです．

```
# 除算'/'を整数除算'//'にした
ans1 = 0 - (1 + 2) + 3 ** 4 * 5 // 6
ans2 = 3 ** 4 // 6 * 5 + 0 - (2 + 1)

print(ans1 == ans2,'. ans1=',ans1,'. ans2=',ans2)
```

を実行すると

```
False . ans1= 64 . ans2= 62
```

となります．

ただし，累乗については並んだ時に左からでなく右からです．（数学的にもこれは常識）

```
4 ** 3 ** 2
```

は 4^{3^2} ですが，これは左から計算した $(4*4*4)^2$ ではなく，右から計算した $4^{(3*3)}$ ですよね．

6.2. 比較演算子 (値と値の関係が正ければ True，間違っていれば False)

優先順位的には，算術演算子より低い（後に計算すべき）です．

```
97 == 97
"a" == "a"
97 != 2
"b" > "a"
"2" > "10"
```

>, >=, ==, <=, <, !=　文字列どうしの比較は，辞書順と考える．つまり，'b' と言う文字列は'a' より後に出てくる．'2' という文字列は'10' よりあとにでてくる．

- type が異なる値の比較

  数値どうしの `int` と `float` は比較できる．数値と文字列の比較はできない．が`!=`(異なる）だけは使える．

```
97 == "a"
```

'a' は文字コードで 97 だが，だからといって 97 と同値ではない．type(97) は int で type('a') が str で型が異なる．

```
97 != 'a'
```

is , is not（後日，「リスト」を学習する際にまた説明します） python には，==と!=と似たものに is と is not‘というものがある．しかし，これは，記憶される場所が同じかどうかを調べている．

```
>>> x = 97, y = 97
>>> x == y
True
>>> x is y
True
>>> id(x) == id(y)
True

>>> x = [97]
>>> y = [97]
>>> x == y
True
>>> x is y
False
>>> id(x) == id(y)
False
```

6.2. 論理演算子

優先順位は，比較演算子よりさらに低いです（後に計算する）
and, or, not

前提

- 0 と 0.0…0 は False, 0 以外は True
- 空文字'' は False, 空っぽでなければ True

and は，「かつ」

- どちらも True のとき True, 論理積
- 左側を見て **false** なら右側を見ずに左側を出力．左側が **True** なら右側を出力

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
```

```
>>> 0 and 10
0
>>> 1 and 10
10
>>> 10 and 1
1
>>> 10 and 0.0
0.0
```

or は「または」

- どちらも False のとき False，論理和
- 左側を見て **True** なら右側を見ずに左側を出力．左側が **False** なら右側を出力

```
>>> True or True
True
>>> True or False
True
>>> False or False
False

>>> 10 or 1
10
>>> 10 or 0
10
>>> 0 or 10.0
10.0
```

```
>>> not True
False
>>> not False
True

>>> not 1
False
>>> not 0
True
>>> not 10
False
>>> not 0.0
True
```

not はブール反転

6.3. ビット演算子

&, |,~,^ はビット毎の and,or,not,xor

```
>>> 3 and 4
4
>>> 4 and 3
3
>>> 3 & 4 # 011 と 100 のビット毎の and は，000 (0)
0

>>> 3 or 4
3
>>> 4 or 3
4
>>> 3 | 4 # 011 と 100 のビット毎の or は，111 (7)
7

>>> not 2
False
```