Pythonの基礎知識

@nc30mtd

pythonコードの書き方と実行方法

- 新規ファイルを作成し，テキストエディタでテキストファイルを編集する（`sample.py` とする）
- 端末から下記のコマンドを実行する

```
$ python3 sample.py
```

※インタラクティブモードで実行する方法もある

pythonの学習のみを目的とする場合はjupitorNotebookがおすすめ．ブラウザからpythonの実装を行うことができる．また，googleアカウントを持っている場合にはColaboratoryが無料で利用できる．（一部制限がある）Linuxインフラの話とpythonの実行に関する話の切り分けができるので，初学者はこちらがおすすめ．

- https://colab.research.google.com/

pythonコードを実装するにあたって「おまじない」で覚えてもらいたいこと

プログラムのエントリポイント（開始地点）について

明示的に下記の宣言を行うと，その行をエントリポイントとしてプログラムを実行する

```
if __name__ == '__main__':
```

上記の宣言を行わなかった場合，一般的なプログラム同様ソースコードの上から順に実行される.
※ただし関数が定義されている場合は読み飛ばす.

画面上に処理結果を表示するprint関数

下記のようにprint()関数を利用すると，引数（後で述べる括弧の中身の変数のこと）の内容を表示する.
定数や変数を引数に取ることができる.

```
# 文字列を表示する場合
text = "日本語"
print(text)
print("日本語")

# 数値を表示する場合
x = 2
print(2) # 値を直接指定する場合
print(x) # 変数を指定する場合
print(x + 2) # 演算結果を表示する場合
print(x, 2) # カンマで区切って複数表示も可能
```

コメント

Python ではコードの中に# が出現すると，それ以降の行の終わりまでがコメントになります.
コメントは行頭からも、行の途中からでも始めることができます.
プログラムの実行時には、コメントは無視されます.
行頭や行内に「#」を追記してコメントを設定し，プログラムの実行結果に影響を与えないようにする行為を「コメントアウト」と呼ぶ場合があります.

```
x = 1
y = 0
# y = x + 1

print(y)

結果：
0
```

上記の例では 3行目に `y=x+1` と `y` に和を代入する式が存在するが，コメントアウトされているため実行結果には影響しない.

変数

プログラミング言語における変数とは，値に対して名前をつける仕組みです．
名前のついた容器に変数を入れることで，名前はその容器の中身の値を指し示します．
また，変数（容器）に値を入れることを「代入」と呼びます．

以下は名前が `v` という変数に対して整数値 `90` を代入した例です．

```
v = 90
```

pythonでは変数を宣言し，値を代入した時点でデータ型が決まります．（C/C++/C#やJava等の言語では型を明示的に宣言します）．変数の型を調べたい場合は `type()` 関数を用います．

```
v = 90
type(v)

実行結果：
<class 'int'>
```

関数と引数，返り値

下記の宣言で定義された文のかたまりを「関数」と呼びます.

```
def 関数名(引数.....):
```

関数は何度も行う処理を複数回記載することによりコードの可読性が落ちることを回避する効果があるほか，再利用しやすく機能をまとめるために利用され，ひとまとまりの処理を定義します.
プログラム内に関数を「呼び出し」する文を加えることで機能します.

```
# 呼び出し例
def func01(v):
    val = v + 2
    print(val)

if __name__ == '__main__':
    value1 = 10
    func01(value1)
```

処理を行うにあたって必要な変数を「引数」とよび，関数定義の時点で宣言します.
不要な場合は括弧の中身は空となり，カンマ区切りで複数指定することができます.

処理の結果を関数を呼び出した文で受け取りたい場合は「返り値」を設定します.
返り値を設定する場合は下記のreturn文を関数内に定義します．返り値がない場合は不要です.

```
return 式
```

※「返り値」は「かえりち」と読みます.

算術記号について

数値の演算

- 加算： `+`
- 減算： `-`
- 乗算： `*` ※関数の定義時に引数などに入る例など，演算子とは限らない場合があります
- 除算： `/`
- 割り算の整数部： `//`
- 割り算の剰余（mod）： `%`
- べき乗： `**`

list,tuple,strの演算（連結や繰り返し）

- 連結： `+`
- 繰り返し： `*`

list型,tuple型,dict型

list型

リストは要素が順番に並んだデータ構造です．あらゆる型の格納できます．
要素は順番に並べられ、順序付けには連続した範囲の整数が使用されます．

個々の要素へアクセスするには「インデックス」（順序番号)を指定します．
この際，インデックスは0から始まることに注意します．

```
sample_list = [] # リストの定義

# リストへの要素の追加
v0 = 10
v1 = 11
v1 = 12
sample.append(v0)
sample.append(v1)
sample.append(v2)
print(sample[2]) # sampleの2要素目の値を参照する
```

tuple型

タプルは要素が順番に並んだデータ構造です．あらゆる型の格納できます．
要素は順番に並べられ、順序付けには連続した範囲の整数が使用されます．
個々の要素へアクセスするにはインデックス(順序番号)を指定します．

追加した要素が変更できないという点がリストと異なります．

```
# tupleの定義
sample_tuple = (10, 11, 12)

print(sample_tuple)
```

リストは大括弧 `[]` で定義しましたが，タプルは小括弧 `()` で定義します．

dict(辞書)型

辞書は，キー（key）と値（value）を対応づけるデータです．キーとしては文字列・数値・タプルなどのデータを使うことができます．

```
# dictの定義（空の辞書を宣言する）
sample_dict = {}
sample_dict["first"] = 1
print(sample_dict)
```

```
# dictの定義（値の入った辞書を宣言する）
sample_dict2 = {
  'second' : 2,
  'third'  : 3
}
print(sample_dict2)
```

スライスによる要素の指定

listやtuple型は`[]`を用いて要素の参照を行うことができましたが，複数の要素を参照したい場合にはスライスを用いて参照することが可能です.
スライスはlist型，tuple型および文字列などのデータの一部をインデックスを指定して切り出す参照方法です.

```
# リスト aに対するスライスの指定方法
a[start : stop : step]
```

- start: 開始インデックス
- stop: 開始インデックス
- step: 刻み幅

例）スライスによる要素参照例

```
numbers = [0, 10, 20, 30, 40, 50, 60] # リストの定義

print(numbers[0:4]) # 0以上，4未満の要素を参照する
print(numbers[0:4:2]) # 0以上，4未満の要素を2要素おきに参照する
```

数値の演算に関するサンプルコード

整数，実数，文字列

文字列

プログラム内で変数に文字列を代入するなどを行いたい際などは，定義する文字の並びをシングルクォート `'...'` ，もしくはダブルクォート `"..."` で囲んで記述します．英語アルファベットだけではなく日本語をはじめとする多くの言語を取り扱いが可能です．

```
text1 = 'hello'
text2 = '日本'

print(text1)
print(text2)
```

文字列の連結

文字列の連結は演算子を用いて行うことができます.

```
text1 = 'Hello'
text2 = ' '
text3 = 'World'
text = text1 + text2 + text3
print(text)
```

文字列とリスト

文字列はリストとして扱うことができます．インデックスを指定して文字を取り出す事ができます．スライスによる指定も可能です.

```
text = 'Hello World'

print(text[4])
print(text[2:6])
```

条件分岐

if文

プログラム内で条件分岐を行う場合はif文を用いて行います.
以下はif文を用いて大きい数字を返す関数の例です.

```
def cmax(a, b):
    if a > b:
        return a #(1)
    else:
        return b #(2)

val = cmax(4, 6)
print(val)
```

実行結果：

```
6
```

if文の右側には条件分岐を行うため，比較演算子を用いた条件を記載します.
比較演算の結果が真であれば(1)，偽であれば(2)の文が実行されます.

if文を制御する比較演算の例を以下に示します

```
x < y   # x はy より小さい
x <= y  # x はy 以下
x > y   # x はy より大きい
x >= y  # x はy 以上
x == y  # x とy は等しい
x != y  # x とy は等しくない
```

比較演算ではANDやORなどの論理演算を組み合わせることが可能です.

```
x >= 0 and y > 0    # x は0 以上，かつyは0より大きい
x < 0 or y > 0      # x は0より小さい，またはyは0より大きい
```

比較演算の結果はbool型の変数として扱うことも可能で`True`（真），`False`（偽）の値を取ります.

```
>>> a=0
>>> b=10
>>> a > b
False

>>> type(a>b)
<class 'bool'>

>>> ret = a > b
>>> print(ret)
False

>>> if(ret):
...     print("a")
... else:
...     print("b")
...
b
```

条件式が複数ある場合

複数の条件で条件分岐を行うことも可能です.

```
a = -1

if(a < 0):
  print("負")
elif(a == 0):
  print("零")
else:
  print("正")
```

繰り返し

プログラム中で繰り返し処理を行いたい場合はfor文またはwhile文を活用します.

for文による繰り返し

for文の活用方法には下記の２通りの方法がある

- 文字列・リスト・辞書などの各要素に対して処理を行う場合
- 指定回数だけ実行する場合

文字列・リスト・辞書などの各要素に対して処理を行う場合のfor文テンプレートは以下の通り.

```
for 変数 in 文字列・リスト・辞書など:
    実行文
```

実行時, 変数 に文字列・リスト・辞書の要素が一つづつ順番に取り出されながら 実行文 が実行される.

また，指定回数だけ実行する場合は以下の通り．

```
for 変数 in range(カウンタ下限値，カウンタ上限値):
    実行文
```

実行文は `カウンタ下限 = 変数` , から `変数 < カウンタ上限` まで `カウンタ上限 - カウンタ下限 -` 回実行される．
現在の繰り返し回数は `変数` に代入される．
`カウンタ下限値` と `カウンタ上限値` には整数値を指定する．

`変数 i` には0〜9までの数値が順番に代入されながら繰り返し処理により `print()` が呼ばれる．

例）リスト `words` から順番に1要素づつ取り出す.

```
words = ['apple', 'cat', 'mouse']
for w in words:
  print(w, len(w))
```

例）同一の処理を１０回繰り返す.

```
for i in range(0, 10):
  print(i)
```

例）リスト `words` から順番に1要素づつ取り出す.　（インデックスを使って参照する方法）

```
words = ['apple', 'cat', 'mouse']
for i in range(0, len(words)):
  w = words[i]
  print(w, len(w))
```

while文による繰り返し

while文では `while` の後の条件式が `False` となるまで実行文グループを繰り返します.

```
while(条件式):
  実行文
```

例）while文による繰り返し例

```
idx = -5
while idx < 0:
    print(idx)
    idx = idx + 1
```

break文

break文は，for文もしくはwhile文の実行文グループで利用可能です.
break文は実行中のプログラムで最も内側の繰り返し処理を中断し、そのループを終了させる目的で利用されます

例）break文による繰り返し例

```
idx = -5
while True: # 条件式が常にTrueなので終了しない（無限ループ）
  print(idx)
  idx = idx + 1

  if idx == 0:
    break # 終了条件に合致すればループを抜ける
```

pass文

pythonでは空の実行文は許されないので利用する制御文.

例えば下記の条件分岐を用いたコードはエラーが返る.

```
a = -1

if(a < 0):
  # IndentationError: expected an indented block
elif(a == 0):
  print("零")
else:
  print("正")
```

上記のコードの目的としては`a`の値が負の場合は処理を行わずにプログラムの実行を継続することであるが，実際に実行するとエラーが生じる.

この場合，何も実行しないpass文を利用することでエラーを回避することができる.

```
a = -1

if(a < 0):
  pass
elif(a == 0):
  print("零")
else:
  print("正")
```

練習問題 1

下記のような計測データをリスト型変数 `a` に読み込んだ.
計測データは平面上の座標値を示しており，X座標とY座標が交互に定義されている.

```
a= [10, -2, 9, -3, 8, -4, 7, -3]

x = []
y = []
```

これをX座標配列 `x` とY座標配列 `Y` に分離する配列を作る.

補足

リストの応用：多重リスト

リスト型にはリスト型変数を追加することも可能です

```
rsample = [[1, 2, 3], [10, 20, 30], ['a', 'b', 'c']]
```

辞書型も追加できます

```
a = {
  'first': 1
}
b = {
  'a', 'ALFA',
  'b', 'BRAVO'
}
rdict = [a, b]
```

リストの応用：関数

`len()` リストの要素数を返す

`min()` リストの最大値を返す

`max()` リストの最小値を返す

`sum()` リストの総和を返す

`sorted()` 並べ替え

```
numbers = [0, 1, 2, 3, 4, 5]
len(numbers)
min(numbers)
max(numbers)
sum(numbers)
sorted(numbers)
```

リストの応用：メソッド

`list.index()` 指定した要素のインデックス取得

`list.count()` 指定した要素のリスト内出現数

`list.sort()` 並べ替え

```
numbers = [0, 1, 2, 3, 4, 5]
numbers.index(3)
numbers.count(3)
numbers.sort()
numbers.sort(reverse = True) #逆順並べ替え
```

`list.append()` 要素の追加

`list.extend()` リストの連結

```
numbers1 = [0, 1, 2, 3, 4]
numbers2 = [-5, -4, -3, -2, -1]

numbers1.append(5)
print(numbers1)

numbers2.extend(numbers1)
print(numbers1)
print(numbers2)
```

`list.insert()` 要素の挿入

`list.remove()` 要素の削除（要素の値で指定）

`list.pop()` 要素の削除（インデックスで指定）

`list.reverse()` 逆順リストの取得

```
numbers = [0, 1, 2, 3, 4]

numbers.insert(0, -1)
print(numbers)

numbers.remove(4) # 存在しない要素を削除すると例外がでる
print(numbers)
numbers.pop(4)  # 存在しないインデックスを指定すると例外がでる
print(numbers)

numbers.reverse()
print(numbers)
```

`list.copy()` リストの複製

リストは変数に代入した場合は「参照渡し」されるため，代入後のリスト変数を編集するともとのリストも変更される．

一方で `list.copy()` を用いた場合は同じ要素を持つ別の配列を新たに確保するので，編集を行っても元の配列は変更されない．

```
numbers = [0, 10, 20, 30, 40, 50]

numbers_copy_1 = numbers
numbers_copy_2 = numbers.copy()



numbers_copy_1.remove(0)

print(numbers)
print(numbers_copy_1)
print(numbers_copy_2)
```

`del` リストまたは要素の削除（インデックスで指定）

```
numbers = [0, 10, 20, 30, 40, 50]

del numbers[2]
print(numbers)

del numbers
print(numbers)
```

リストを削除したのちに参照を行うと例外メッセージが出る.

```
NameError                                 Traceback (most recent call last)
<ipython-input-7-43235806bb58> in <module>()
      4 del numbers
      5
---
<!--_class: normal-->-> 6 print(numbers)

NameError: name 'numbers' is not defined
```