python基礎（中級者向け）

新海典夫（理研　革新知能統合研究センター（AIP））

※中級者向けということで、既にご存知の事項も多く、退屈に感じられる方も多いかと思います。次の段階への準備運動ということで、しばしお付き合い頂けましたら幸いです。

※頑張ってタイピングしよう！

準備

ターミナルを起動します。
以下のコマンドを実行して下さい。

$ python3

以下のような画面になるかと思います。

```
Python 3.6.5 (default, Apr  1 2018, 05:46:30)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

この状態をidleと言います。

以下、しばらくidleで作業をしていきます。

変数

変数には”=”で代入を行います。また、変数には型宣言は必要ありません。
(尚、文字列において、シングルクオーテーションとダブルクオーテーションはどちらも使えます)

```
>>> val_int = 1
>>> val_int
1
>>> val_float = 1.1
>>> val_float
1.1
>>> val_str = "atgc"
```

```
>>> val_str
'atgc'
```

以上のように、数値も文字列も入ります。

type()　で、型を確認しておきましょう。

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

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

>>> type(val_str)
<class 'str'>
```

複合データ型

リスト、タプル、辞書といった複数の値の構成が可能なデータ型があります。確認しておきましょう。

リスト

リストの作成

```
>>> list1 = [1, 3, 5, 7, 9, 11]
>>> list1
[1, 3, 5, 7, 9, 11]
```

これでリスト”list1”が作成されました。

尚、空のリストを作成することもできます。

```
>>> list2 = []
>>> list2
[]
```

複数の型が混在するリストの作成も可能です。

```
>>> list3 = ["chr1", 1358, 2358]
>>> list3
```

['chr1', 1358, 2358]

リスト自体も型ですので、入れ子構造も可能になります。

```
>>> list4 = [1, 3, list3]
>>> list4
[1, 3, ['chr1', 1358, 2358]]
```

要素の取得

インデックスを指定することで「何番目かの要素」にアクセスすることができます。

```
>>> list1[0]
1
>>> list1[1]
3
```

マイナスの値をしていすることで「後ろから」アクセスすることもできます。

```
>>> list1[-1]
11
>>> list1[-2]
9
```

存在しない値にアクセスしようとするとエラーになります。

```
>>> list2[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

尚、len()にて、リストの要素数自体を取得することもできます。

```
>>> len(list1)
6
```

スライス

リスト名[開始値 : 停止値 : 間隔]とすることで指定インデックス範囲のリストを取り出すこともできます。
#開始点はゼロです。

```
>>> list1[1:4]
[3, 5, 7]
>>> list1[:2]
[1, 3]
>>> list1[2:]
[5, 7, 9, 11]

>>> list1[0:5:2]
[1, 5, 9]
```

間隔をマイナスにすることで逆順とすることもできます。

```
>>> list1[::-1]
[11, 9, 7, 5, 3, 1]
```

連結

リストとリストを結合させ、新しいリストを作ることもできます。

```
>>> list3
['chr1', 1358, 2358]
>>> list5 = ["atgc", 1]
>>> list5
['atgc', 1]
>>> list6 = list3 + list5
>>> list6
['chr1', 1358, 2358, 'atgc', 1]
```

要素の入れ替え

リストは可変（作成後に変更可能）です。
インデックスを指定して、そこだけ要素の変更、といったことができます。

```
>>> list1
[1, 3, 5, 7, 9, 11]
>>> list1[3]
7
>>> list1[3] = "edited"
>>> list1[3]
'edited'
>>> list1
[1, 3, 5, 'edited', 9, 11]
```

リスト名.append(値)で末尾への値の追加ができます。

```
>>> list1
[1, 3, 5, 'edited', 9, 11]
>>> list1.append("append_data")
>>> list1
[1, 3, 5, 'edited', 9, 11, 'append_data']
```

リスト名.insert(位置, 値)で指定位置への挿入もできます。

```
>>> list1.insert(3, "insert_data")
>>> list1
[1, 3, 5, 'insert_data', 'edited', 9, 11, 'append_data']
```

ソート

リストはソートすることができます。

```
>>> list7 = sorted(list6)
>>> list7
[1, 2, 3, 4, 5, 6, 8]
```

この時、元のリストは変更されていません。

```
>>> list6
[3, 5, 8, 6, 4, 2, 1]
```

ソートは逆順でも行えます。

```
>>> list8 = sorted(list6, reverse=True)
>>> list8
[8, 6, 5, 4, 3, 2, 1]
```

尚、元のリスト自体をソートするときには、sort()を使います。

```
>>> list6.sort()
>>> list6
[1, 2, 3, 4, 5, 6, 8]
```

keyを使ったソート

keyパラメータを用いて、様々な条件でのソートが可能になります。

```
# 絶対値でソート
>>> list9 =  [-3, 1, -5, 4, 2, -6]
>>> list_abs = sorted(list9, key=abs)
>>> list_abs
[1, 2, -3, 4, -5, -6]
```

```
# 文字列の長さで降順でソート
list10 = ["shizuoka", "tokyo", "mie", "hokkaido", "osaka"]
>>> list_len = sorted(list10, key=len, reverse=True)
>>> list_len
['shizuoka', 'hokkaido', 'tokyo', 'osaka', 'mie']
```

```
#独自関数でソート
```

関数 func(x)を作成します。

```
>>> def func(x):
...   tmp = x.split("_")[-1]
...   return int(tmp)
...
```

これは、”gene_1”といった文字列があった場合、最後の数字だけを取り出して、整数として返すという機能を持っています。

```
>>> func("gene_1")
1
```

これを組み入れてソートしてみます。

```
>>> list11 = ["gene_1", "gene_10", "gene_11", "gene_19", "gene_2", "gene_20", "gene_21", "gene_29", "gene_3", "gene_30"]
>>> list11_func = sorted(list11, key=func)
>>> list11_func
['gene_1', 'gene_2', 'gene_3', 'gene_10', 'gene_11', 'gene_19', 'gene_20', 'gene_21', 'gene_29', 'gene_30']
```

無名関数でソート

上のソートを、lambda式（関数を名前を定めず使う手法）を使うことで関数定義無しに同様にワンライナーで書くこともできます。
lambda 引数: 式
という書き方で書きます。
それを組み入れると以下の通りになります。

```
>>> list11_lambda = sorted(list11, key=lambda x: int(x.split("_")[-1]))
>>> list11_lambda
['gene_1', 'gene_2', 'gene_3', 'gene_10', 'gene_11', 'gene_19', 'gene_20', 'gene_21',
'gene_29', 'gene_30']
```

タプル

タプルとは、immutable(不変)なリストです。
x = (要素1, 要素2, 要素3)
といったかたちで作成します。

```
>>> tuple1 = ("gene1", "chr8", 1, 2, 3)
>>> tuple1
('gene1', 'chr8', 1, 2, 3)
```

要素へのアクセスや、スライスはリスト同様に可能です。

```
>>> tuple1[0]
'gene1'
>>> tuple1[3]
2
>>> tuple1[0:1]
('gene1',)
>>> tuple1[0:3]
('gene1', 'chr8', 1)
```

ただしタプルはimmutableなので、要素の追加や変更等はできません。
リストでは行えたような以下の操作はいずれもエラーになります。

```
>>> tuple1[0] = "gene2"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> tuple1.append("4")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

また、タプルとリストは相互変換（新たな生成）が可能です。

```
>>> tuple1
('gene1', 'chr8', 1, 2, 3)
>>> list_from_tuple1 = list(tuple1)
>>> list_from_tuple1
['gene1', 'chr8', 1, 2, 3]
>>> tuple(list_from_tuple1)
('gene1', 'chr8', 1, 2, 3)
```

リストの凍結↔解凍といった関係になっている、とも言えます。

参考：タプルについて

タプルは処理がリストより高速になります。
タプルはデータが不変である為、データに『書き込み保護』的要素を持たせる意味もあります。

辞書型（ディクショナリ）

キーと値、というデータの組を保存していく仕組みです。
ディクショナリ名　＝　{キー：値, キー:値…}
と書いていきます。

```
>>> gene_set1 = {"chr":1, "pos":324, "ref":"c" , "var":"t"}
```

これは適当に書いたものですが、例えば、染色体１番、324塩基目、本来の塩基はc、変異後の延期はt、となります。

```
>>> gene_set1["chr"]
1
```

とすることで、gene_set1内で、染色体は？として1、という値が取り出せる訳です。

セット（集合）

データの集合を扱う仕組みです。データ内に順序はありません。

```
>>> set1 = {"gene_1", "gene_3", "gene_8"}
>>> set1
{'gene_3', 'gene_1', 'gene_8'}

>>> set2 = {1 , 2, 3}
```

```
>>> set2
{1, 2, 3}
```

セットには要素の追加や削除ができます。

```
>>> set1
{'gene_3', 'gene_1', 'gene_8'}

>>> set1.add(12)
>>> set1
{12, 'gene_3', 'gene_1', 'gene_8'}

>>> set1.remove("gene_1")
>>> set1
{12, 'gene_3', 'gene_8'}
```

また、集合演算（和、積、差、対称差）も可能になっています

和
```
>>> set3 | set4
{1, 2, 3, 4, 5, 6, 7}
```

積（共通部分）
```
>>> set3 & set4
{3, 4, 5}
```

差
```
>>> set3 - set4
{1, 2}
```

対称差（どちらから一方にのみある値）
```
>>> set3 ^ set4
{1, 2, 6, 7}
```

尚、リストは集合に変換できます。

```
>>> list_s = (1, 2, 3)
>>> list_s
(1, 2, 3)
>>> set(list_s)
{1, 2, 3}
```

尚、集合では同じ値の重複はありません。その為、値の重複のあるリストでは、こんな風になります。

```
>>> list_s2 = (1, 1, 2, 3, 4, 5)
```

```
>>> list_s2
(1, 1, 2, 3, 4, 5)
>>> set(list_s2)
{1, 2, 3, 4, 5}
```

リストでは重複した値が、集合では一つにまとまりました。

ソースを作成してのpython実行

ここから先は、スクリプトを作成してのpythonの実行に入ります。

テスト用のフォルダを作りましょう。次からここで作業していきます。

```
$ cd ~
$ mkdir python_test1
$ cd python_test1/
$
```

エディタで次のようなファイルを作成して、ファイル名”test1.py”として保存します。

```
print("test1")
print("test2")
```

（記入して保存）

実行してみます。python3 スクリプト名　で実行できます。

```
$ python3 test1.py
test1
test2
```

尚、この出力は直接ファイルに書き出すこともできます。

```
$ python3 test1.py > test1.txt
```

内容を書き出す、linuxのcatコマンドで見てみます。

```
$ cat test1.txt
test1
```

```
test2
```

先ほどと同じ内容がファイルに保存されていたことがわかります。
エディタで開いて中身を確認することもできます。

制御構文

if文、for文といった制御構文について確認しておきます。

if文

エディタで次のファイルを作成して、if_test.pyとして保存して下さい。

```
gene_set1 = {"chr":1, "pos":324, "ref":"c" , "var":"t"}

if gene_set1["pos"] >= 300:
    print("above 300")
```

（これで保存）

$ python3 if_test.py
above 300

辞書型gene_set1のposが300を超えていたらover 300と出力する、それだけのものです。
gene_set1のposの数値を、例えば250に書き換えて再度実行してみると、何も表示されない、というのが分かるかと思います。

先ほどのソースを以下のように少し書き加えてみて下さい。

```
x = 300

if x >= 300:
    print("above 300")

elif x >= 200:
    print ("above 200")

else :
    print("under 200")
```

xの値を、色々と変えて実行してみてください。

(x=300)
$ python3 if_test.py
above 300

(x=200)
$ python3 if_test.py
above 200

(x=150)
$ python3 if_test.py
under 200

論理演算

先ほどは、シンプルにxの値で条件分けをしていました。
ですが、例えば x >=300 で y >= 200なら…といった複数の条件を組み合わせることも考えられます。論理演算によってこのような操作が可能になります。
次のようなスクリプトを作成して、if_test2.pyと保存してみて下さい。

```
x = 350
y = 250

if x >= 300 and y > 200 :
    print("success")

elif x >= 300 or y > 200 :
    print ("half success")
```

条件1 and 条件2、とすることで、両方ともTrueの場合に条件を満たしたと判断されます。
条件1 or 条件2、とすることで、片方がTrueの場合に条件を満たしたと判断されます。

xとyの値を変化させて試してみて下さい。

さらにnot (否定)もあります。上のスクリプトを少し書き換えてみて下さい。

```
x = 150
y = 150

if not(x>=300) and not(y>=200) :
```

```
    print("success")
```

こうすることでx>=300「ではなく」、かつ、「y>=200」ではない時のみTrue、という判断がされることになります。

for文

for 変数 in データの並び（を生成する仕組み）:
　　処理１
　　処理２

とすることで、データの並び分だけ反復処理を行うものです。

たとえば、リストはこの並びとして使うことができます。
以下のようなスクリプトを書き、for_test.pyとして保存してみて下さい。

```
list1 = ["gene_01", "gene_02", "gene_03"]

for gene in list1:
    print(gene," is tested")
```

実行結果：

$python3 for_test.py
gene_01 is tested
gene_02 is tested
gene_03 is tested
$

高度なリスト：リスト内包表記

for文等を抑えたところで、リスト内包表記についても触れておきます。
リスト内包表記は、リストの各要素に関数を適用することで、新しいリストを取得する、特殊な表記方法です。

[式 for 変数 in リスト if 条件]

といったかたちで書きます。

＊少しだけidle入力に戻ります。

```
>>> list_test = [1, 9, 8, 4]
>>> list_test
[1, 9, 8, 4]

>>> list_comp = [x * 2 for x in list_test]
>>> list_comp
[2, 18, 16, 8]
```

左から見ていきます。
for x in list_test
とあります。
for 変数 in リストのかたちで、list_testからひとつづつ要素を取り出します。
先ほどのfor文と同じ形です。

次に、x *2 です。
取り出した、それぞれを*2しています。

さらに条件を付け加えてみます。

```
>>> list_comp = [x * 2 for x in list_test if x!=8]
>>> list_comp
[2, 18, 8]
```

x!=8 (xが8でないもの)の条件をプラスしました。
その為、
[1, 9, 8, 4]のうち、[1, 9, 4]のみが条件に合う数値となり、その二倍が新たなリストとして生成された訳です。

コマンドライン引数

ところで、先ほどのif文のスクリプト（if_test2.py）ですが、xとyの値はあらかじめ決まっていました。スクリプトの実行時にこの値を決めることが出来たら、色々と便利かとは思います。その機能として、コマンドライン引数というものが用意されています。少しだけ紹介します。先ほどのif_test2.pyを少しだけ以下のように書き換えて、argv_test.pyとして下さい。

```
import sys
```

```
x = int(sys.argv[1])
y = int(sys.argv[2])

if x >= 300 and y > 200 :
    print("success")

elif x >= 300 or y > 200 :
    print ("half success")
```

以下のように実行してみて下さい

$python3 arg_test.py 350 250
success

$python3 arg_test.py 350 150
half success

こうすることで、スクリプトの後に書かれた二つの値が、それぞれ、

sys.argv[1]
sys.argv[2]

としてスクリプト内の値として取り込まれていることになります。
（もっとも、取り込まれた値は文字列型（str型）なので、整数値に変換した上でxとyに代入してしますが。）

このように、スクリプト実行時に引数を渡す仕組みもあります。
(尚、より高度な引数渡しの仕組みとしてargparseというものもあります。)

関数

他の多くのプログラミング言語と同様に、pythonには、一定の処理をまとめて記述するものとして、関数という機能が用意されています。確認しておきましょう。

def 関数名(引数リスト):
　　処理

こういったかたちで関数は記述します。
次のようなスクリプトを書いてみて下さい。func_test1.pyとして保存しましょう。

```
def hello(x):
    print("hello ",x)

hello("Jannet")
```

実行してみます。

$python3 func_test1.py
hello Jannet

上の例では関数の()内に引数を渡してますが、キーワードで指定する、というやり方もあります。

```
def hello(name, time):
    print("hello ",name)
    print("clock:", time)

hello(name = "Jannet", time = 3)
```

$python3 func_test1.py
hello Jannet
clock: 3

戻り値にはreturnを用います。func_test2.pyとして保存してみましょう。

```
def check_name():
    return "Jannet"

print("hello ", check_name())
```

$python3 func_test2.py
hello Jannet

check_name関数は、文字列”Jannet”を戻り値としています。
print文の中でその関数が呼び出されている訳です。
returnでは、複数の値を戻すこともできます。

```
def check_name():
    return "Jannet" , "Andy"

x, y = check_name()

print("hello ", x, " and ", y)
```

$python3 func_test2.py
hello Jannet and Andy

（細かい話をすると、この時、戻り値は"Jannet" , "Andy"のタプルとなっています。）

可変長引数についても見ておきます。

以下のスクリプトを作成の上、func_test3.pyとして保存の上、実行してみて下さい。

```
def func_args(*args):
    print("args:", args)

def func_kwargs(**kwargs):
    print("kwargs:", kwargs)

func_args(1, 2, 3, 4, 5)
func_kwargs(a=1, b=2, c=3, d=4, e=5)
```

実行結果：

$python3 func_test3.py
args: (1, 2, 3, 4, 5)
kwargs: {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

まず、*argsです。仮引数（args）の頭にアスタリスクを付けることで、数を定めずに引数をいくつも渡すことが出来ます。渡された引数はリストとして保存され、関数内部で参照出来ます。

次に、**kwargsです。同様に、アスタリスクを二つ付けることで、数を定めずにキーワード引数を渡すことができるのです。渡された引数は辞書型（キーワードと値の組）として保存され、同じように関数内部で参照できる訳です。

クラス

最後にクラスにおいて触れておきます。

クラスとは、「データと機能を組み合わせる方法」を提供するものです。
例えば、ある「車」という「データと機能の組み合わせ(class)」を考えます。
この車には「所有者」「購入何年目」という「データ」があります。
また、この車は、ガソリン（ｘリットル）を入力されるとｘ＊１５ｋｍの走行距離出力するという「機能」があります。
これをクラスで書いてみます。
以下のスクリプトをclass_test1.pyとして作成、保存して下さい。

```
class Car:
    def __init__(self, owner, year):
        self.owner = owner
        self.year = year

    def getYear(self):
        return 2018-self.year

    def mileage(self, gas):
        return gas * 15

car1=Car("Andy", 3)
car2=Car("Cathy", 5)

print("car1_owner:",car1.owner)
print("car2_year:",car2.getYear())
print("car1_mileage:",car1.mileage(10))
```

実行してみます。

$python3 class_test1.py
car1_owner: Andy
car2_year: 2013
car1_mileage: 150

このスクリプト、まず、
class Car:
とあります。
以下のブロックにて、Carというクラスの中身を定めています。
慣例的にクラス名の頭は大文字にします。

そして、ひとまずブロック内を読み飛ばしてください。

car1=Car("Andy", 3)
car2=Car("Cathy", 5)

ここで、car1、car2という実際の、「データと機能の組み合わせ」を二つ作成しています。

この作成されたcar1、car2を専門用語で「インスタンス」と言ったりもします。

そして最後のprint３行です。それぞれ、

car1.ownerでcar1の所有者情報に直接アクセスし、
car2.getYear()でcar2の購入年数を出力させ、

car1.mileage(10)で、「この車でガソリン１０リットルだと何キロ走る？」という演算をしています。

では、次に、Carクラスのブロック内部を見ていきましょう。

まず、def ~ で記載されているそれぞれのブロックを「メソッド」と呼びます。

```
def メソッド名(self, 引数リスト):
        self.変数 = …
        return 式
```

といったかたちで記載します。また、メソッド内のself.変数（self.owner等）をデータ属性と呼びます。

特に、冒頭に記載されている __init__()メソッドは、クラスのインスタンスが生成されるとすぐに呼び出される、となっているものです。

その為、
car1=Car("Andy", 3)
とした段階で、owner, year、二つの引数がそれぞれ"Andy", 3として引き渡され、
self.owner

self.year
という二つのインスタンス変数に代入されます。
同時に、これはインスタンス内部の値となり、外部から参照可能なものとなります。
呼び出すときは、
インスタンス名.owner
インスタンス名.yearとなります。

print("car1_owner:",car1.owner)

はそういう意味です。

一方、
def getYear(self):
def mileage(self, gas):

の二つのメソッドは、「実際に呼び出された時に初めて実行される」ものです。
その為、

print("car2_year:",car2.getYear())
とした段階で、初めてcar2のgetYearメソッドが呼び出されています。
print("car1_mileage:",car1.mileage(10))
とした段階で、初めてcar1のmileageメソッドが、「引数が渡されたうえで」呼び出されているのです。

きわめて簡単ですが、これがメソッドの基本構造となります。

モジュールのimportについて

最後に、少しだけimportについて述べておきます。

pythonでは、既に作成したコードを再利用する仕組みがあります。
実際にどういうものか見ておきます。
先ほどの、class_test1.pyを開いてみて下さい。そして、以下のように

```
class Car:
    def __init__(self, owner, year):
        self.owner = owner
        self.year = year

    def getYear(self):
        return 2018-self.year

    def mileage(self, gas):
        return gas * 15
```

というクラスの定義部分だけ残して、他の部分を削除してしまいましょう。
class_test1_mod.pyとして保存してみます。
これで「モジュール」が一つ準備出来ました。

class_test1_mod.pyを保存したところと同じフォルダで、python3としてidleを起動してみて下さい。

```
>>> import class_test1_mod
```

こうすることで、モジュールclass_test1_mod.pyをインポートすることが出来ました。
idleで次のように入力してみて下さい。

```
>>> car_new = class_test1_mod.Car("Jane", 1)
```

こうすることで、class_test1_mod.pyモジュール内に記載されていた、Carクラスを呼び出すことが出来ます。
その結果、car_newというインスタンスが新たに生成された訳です。

```
>>> car_new.owner
'Jane'
>>> car_new.getYear()
2017
>>> car_new.mileage(15)
225
```

その為、先ほどと同様に、インスタンス内のデータ属性にアクセスしたり、メソッドを呼び出したり、が出来るようになる訳です。

このインポートという作業は、自作のモジュールだけでなく、既存の標準モジュールに対しても行われます。

```
>>> import datetime
```
としてみて下さい。これは時間等を扱う標準モジュールの一つです。
```
>>>datetime.datetime.now()
```

と実行してみて下さい。
現在時刻を取得するものですが、先ほどの、car_new.getYear()
と似たような構造になっているのがお分かりいただけるかと思います。
このようにして、既存のモジュール等をimportして活用していくことになります。

本稿の説明はここまでとなります。お疲れ様でした。

参考文献

Pythonチュートリアル

https://docs.python.jp/3/tutorial/index.html

Dive Into Python 3 日本語版
http://diveintopython3-ja.rdy.jp/index.html

やさしいPython
高橋 麻奈/SBクリエイティブ ISBN-10: 4797396024