(2) Pythonの文法

専門演習B：Pythonによるデータ分析入門

まずは・・・

文字コードの問題

- Python2系列の鬼門（Python3系列ではだいぶマシ）

Python2の内部文字コード：名前だけUnicode

外部とのやりとり：UTF-8（Unicodeの符号化方式の一つ）

文字コードの設定

- ファイルの文字コード指定（ATOMで指定するのはもちろん）
  - ファイルの1行目か2行目に以下のように記述

```
# -*- coding: UTF-8 -*-
```

- ファイル入出力用文字コード指定
  - ファイルの先頭の方で下記のように記述

```
import sys
reload(sys)
sys.setdefaultencoding('utf-8')
```

- 環境変数LC_ALL：これによって自動的に画面出力文字コードが変わる
  - ホームの .bash_profile に以下のように書いておく

```
export LC_ALL=ja_JP.utf-8
```

基礎

シンプルなプログラムの構成

- 全体
  - メイン＋関数
  - 基本は上から下に文を実行
- メイン
  - レベル0の文（基本はインデントがない文）
  - 大抵はファイルの先頭の方か末尾に書かれる
- 関数（後述）
- 文
  - 改行で区別される（末尾にセミコロンは不要）

Pythonの実行

- 対話形式で実行
  - ターミナルで「python」と実行
  - 「>>>」はPythonのコマンドプロンプト（命令待ち状態）

```
$ python
>>> print('hello!')
hello
>>> exit()    # 終了
```

- プログラムの実行
  - ターミナルで「python」の後にプログラム名を指定

```
$ python hello.py
```

コメント

- 正式なコメント：シャープ(#)以降がコメント

```
# this is a line comment
x = 0  # initialize a variable
```

- 複数行にわたるコメント：トリプルクォートでくくる（実は文字列）

```
'''
これは
複数行に
わたる
コメント
'''
```

ライブラリのインポート

- 外部ライブラリを読み込み利用可能にする

```
>>> import math   # 数学ライブラリを利用可能にする
>>> math.e       # math.e を評価
>>> 2.718281828459045
```

- from形式だとライブラリ名を省略できる

```
>>> from math import e   # math.eをインポート
>>> e                    # math.eを評価（mathは省略できる）
```

- 別名での読み込み

```
>>> import math as m   # math を m という名前でインポート
>>> m.e                # math.eを評価
```

print文

- 標準出力（通常は画面）に出力
- 自動で改行が付加
- 書式付き出力（%s：文字列，%d：整数，%f：浮動小数点）

```
>>> print("e=%f" % math.e)
>>> e=2.718282
```

- 複数の変数を同時に出力する場合はタプルを利用

```
>>> print("e=%f log(e)=%f" % (math.e, math.log(math.e)))
>>> e=2.718282 log(e)=1.000000
>>> print("%s %s" % ("Hello","World"))
>>> Hello World
```

変数

- 変数の宣言はない（最初に代入された時に生成される）
- どんな型でも代入できる（下のような利用法も可能）

```
x = 1           # 最初の利用（整数）
x = 'hello'     # 文字列で上書き
```

数値演算

四則演算

演算子	意味	記述例
+	加算	y = 1 + x
-	減算	y = x - 1
*	乗算	y = 5 * x
/	除算	y = x / 2
//	整数の除算	y = x // 2（答えは整数）
%	剰余	y = x % 3
**	累乗	y = x ** 3（xの3乗）

比較演算

演算子	記述例	意味
==	a == 1	a が 1 に等しい
!=	a != b	a と b が等しくない
>	a > 5	a が 5 より大きい
>=	a >= b	a が b 以上である
<	a < b	a が b より小さい
<=	a <= 0	a が 0 以下である

論理演算

- 論理積（～かつ～）：and

```
>>> x = 60
>>> 0 <= x and x <= 100   # xが0以上100以下
>>> True
```

- 論理和（～または～）： or

```
>>> x < 0 or 100 < x      # xが0より小さいまたは100より大きい
>>> False
```

- 否定（～でない）：not

```
>>> not x < 0             # xが0より小さくない
>>> True
```

文字列

文字列（str文字列）

- 通常の文字列
  - シングルクォートかダブルクォートでくくる（どちらでも良い）

```
str1 = 'Hello'
str2 = "World"
```

- 複数行にまたぐ文字列
  - クォート3つでくくる（シングルでもダブルでも良い）

```
str3 = """This
is
Multi-Line
string"""
```

Unicode文字列（Python2のみ）

- マルチバイト文字列（日本語など）
  - 文字列の前に u をつける

```
text_unicode = u'日本語文字列'
```

- unicode文字列とstr文字列の変換
  - unicode文字列 → str文字列

```
text_str = text_unicode.encode('utf-8')
```

  - str文字列 → unicode文字列

```
text_unicode = text_str.decode('utf-8')
```

- 注意
  - 入出力の際はUTF-8で符号化したstr文字列が良い

リスト・タプル・辞書

リストとタプル

- 他言語でいうところの配列
  - ただし異なる型の値を入れることが出来る
- リスト：[ ]
  - 値の代入が可能
  - 要素の追加や削除が可能
- タプル：( )
  - 値の代入や要素の追加・削除はできない
  - 関数の引数や返り値などに利用される

リスト

```
>>> list1 = []                          # 空のリストを作成
>>> list2 = [1, 3.14, "hello"]          # 値の入ったリストを作成
>>> list2[1]                            # 2番目の要素を評価
>>> 3.14
>>> list2[-1]                           # 後ろから1番目の要素を評価
>>> "hello"
>>> list2.append(3)                     # リストの最後に要素を追加
>>> list2                               # 確認
>>> [1, 3.14, 'hello', 3]
>>> list2[3] = "world"                  # 4番目に値を代入
>>> list2                               # 確認
>>> [1, 3.14, 'hello', 'world']
```

タプル

```
>>> tuple1 = ()                          # 空のタプルを作成
>>> tuple2 = (1, 3.14, "hello")          # 値の入ったタプルを作成
>>> tuple2[1]                            # 2番目の要素を評価
>>> 3.14
>>> tuple2[-1]                           # 後ろから1番目の要素を評価
>>> "hello"
>>> tuple2.append(3)                     # タプルの最後に要素を追加
>>> Traceback (most recent call last):   <--- エラー
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> tuple2[1] = 3.1415                   # タプルの要素に代入
>>> Traceback (most recent call last):   <--- エラー
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

辞書（ディクショナリ）

- 他言語でいうところの連想配列
  - 数値以外に文字列やタプルをキーに値を格納できる

```
>>> dic1 = {}                          #  空の辞書を作成
>>> dic2 = {'a':1, 'b':2}              #  値の入った辞書を作成
>>> dic2['a']                          #  'a'をキーとして値を評価
>>> 1
>>> dic2['c'] = 3                      #  新たなキーと値を追加
>>> dic2.keys()                        #  キーの一覧
>>> ['a', 'c', 'b']
>>> dic2.values()                      #  値の一覧
>>> [1, 3, 2]
>>> del dic2['a']                      #  キーの削除
```

制御構造

条件分岐(1)

- if文

```
if 条件:
    print('true.')
    x += 1
```

同じインデントがまとめて実行される（ブロック）

- if～else文

```
if 条件:
    print('true.')
    x += 1
else:
    print('false.')
    x -= 1
```

条件分岐(2)

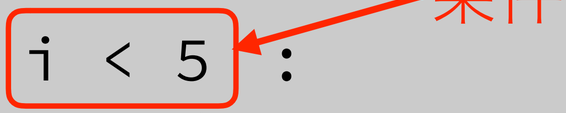
- if～elif文

```
if 条件1:
    print('the condition1 is true.')
elif 条件2:
    print('the condition2 is true.')
else:
    print('both conditions are false.')
```

繰り返し

- while文：条件が満たされている間繰り返す

```
i = 0
while i < 5 :
      print('i is still less than 5.')
      i += 1
```



- for文：リスト・タプル等から1つずつ取り出す

```
for i in [0,1,2,3,4,5]:
      print('i = %d'%i)
```

forを使った定型的な文

多数の繰り返し

```
for i in xrange(10000):
     print('i = %d'%i)
```

辞書の中身を列挙

```
for key in dic.keys():
     value = dic[key]
     print('key:%s value:%s'%(key, value))
```

条件に合う値の組をリストにする

```
list = [(p.x, p.y) for p in locations if p.valid == True]
```

関数

基本的な関数

- 関数定義は def で始める
- 値は return で返す

```
def add(num1, num2):
    result = num1 + num2
    return result
```

- 収まりきらなかった引数はタプルとして *付き引数で渡される

```
def f(a, *args):
    print('a = %s'%a)
    i = 0
    for v in args:
        print('args[%d] = %s'%(i,v))
        i += 1
```

キーワード引数を持つ関数

- キーワード引数は辞書として **付き引数に渡される

```
>>> def f(**kwargs):
...     for key in kwargs.keys():
...             value = kwargs[key]
...             print('key:%s\tvalue:%s'%(key,value))
...
>>> f(a=1, b=2, three=3)
>>> key:a value:1
key:b value:2
key:three value:3
>>>
```