クレジット:
UTokyo Online Education Education コンピュータシステム概論 2018 小林克志

東京大学
THE UNIVERSITY OF TOKYO

CC BY NC ND

Python の基本

この資料は The Python Tutorial (https://docs.python.org/3.6/tutorial/index.html#the-python-tutorial) (日本語版 (https://docs.python.jp/3/tutorial/)) および Python for Data Analysis:Wrangling with Pandas, Numpy and IPython (http://shop.oreilly.com/product/0636920050896.do)を参考に作成した。
このチュートリアルでは他のプログラミング言語を習得している方を想定し、Python の特徴などを説明する。

オブジェクト

Python の重要な特徴に、一貫したオブジェクトモデルが挙げられる。 全ての数、文字列、データ構造、関数、クラス、モジュールなどは Python でオブジェクトとして参照される「箱」として扱われる。
全てのオブジェクトは（文字列、関数といった）型と内部データを持ち、これによって Python は言語としての柔軟性を実現している、 例えば、全ての関数が他のオブジェクトと同様に扱うことができる、例えば関数を変数に代入することもできる。

近代的な手続き型言語はほぼオブジェクト指向を取り入れており Python の特徴とはもはや言えないかもしれない。

インデント

Python では行頭の字下げ（インデント）でプログラムを構造化している。
同じインデントレベルの一連のプログラム文をコードブロックと呼び、インデントによる構造化は Python の特徴である。
他方、C, Java, Ruby といった他の言語では構造化にはおもに括弧 `{}` 、（開始から終了までがコードブロック）が利用されている。

インデントはタブ、スペース(空白)いずれも使えるが、Python 標準ライブラリでは スペース 4文字を標準 (http://pep8-ja.readthedocs.io/ja/latest/)とするスタイル(書式)を採用している。
Jupyter の標準もこれを採っており講義でもこの書式を利用する。 セルでタブを入力すれば勝手に変換されるのであまり気にする必要はない。

プログラムの例を使って説明する。
変数 `x, y` の符号を印字するプログラムは、`if` 文と `print()` 関数で以下のように書ける:

In [ ]:
```
x = 1
y = 1
if x > 0 :
    print ("x has the positive sign, but y is unknown")
    if y > 0 :
        print ("Both 'x' and 'y' have the positive.")

print("Sign check is done.")
```


3行目の `if` 文の行末の`:` はコードブロックの先頭を意味し、これに続く同じ字数だけインデントされた行の続く範囲までコードブロックは続く。そのコードブロックが直前の `if` 行の制御対象となる。

5 行目から二重の `if` 文となり、この制御範囲は、二重にインデントされた 6 行目の `print()`関数のみのコードブロックとなる。

8 行目のインデントされていない `print()`行は、いずれの `if` 文の制御範囲ではなく、（`x,y` の符号にかかわらず）かならず実行される。

セミコロン

Python の文はセミコロン`;`で終わらせる必要はない。
ただし、複数のプログラム文を一行にまとめるためセミコロンを使うこともできる。

In [ ]:
```
print("The"); print("quick"); print ("brown") ; print ("fox"); pr
int("jumps")
```

コメント

ハッシュマーク、または pound sign、 #に続く文字は Python では全て無視されます。 このため、#はプログラム中にコメントを書くために使われます。 そして、コメントはプログラム行の後ろに置くこともできる。
付け加えると、コーディング中には一部のプログラム機能を無効化（コメントアウト）するためにも使われる。

以下のコードでは、冗長な `print()` 文をコメントアウトしている:

In [ ]:
```
#
j = 0
for i in range(10):
    j += i
    # print(i)   # i を出力
    # print(j)   # j の途中結果を出力

print(j)           # 結果を出力
```

複数行のコメント

著作権表示のようにコメントが複数行にわたる場合は、

- 全ての行頭に # を置きます
- 全ての行をそれぞれ引用符で囲む
- ３つの引用符`'''`、あるいは二重引用符`"""`でブロックを囲む
  下２つの方法は文字列を表現する手段であるが、Python は文字列のみのプログラム文は無視する仕様を利用している

以下は Python プログラム冒頭に書く著作権表示の例:

In [ ]:
```
#
# Copyright (C) 2018 Katsushi Kobayashi
# All Rights Reserved.
#
print("Longevity and prosperity")
```

関数、オブジェクトメソッドの呼び出し

関数、オブジェクトメソッドは()内に引数を、複数の場合はコンマ,区切りで与えて呼び出す。

Python の引数の与え方

Python では引数は以下の3 種類の与え方が可能である。

1. 順番: `f(10, 100, "foo", "bar")`
2. キーワード: `f(keyword1 = 100, keyword3 = "foo", keyword2 = 100, keyword4 = "bar")`
3. 1.,2, の混在: `f(10, 100, keyword4 = "bar",keyword3 = "foo")`

付け加えると、関数・オブジェクトメソッド側で引数の既定値が設定されていれば、呼び出し時にその引数を省略できる。
キーワードと、省略可能な引数は Pytgon の特徴の一つで、オプションの多い描画関係の関数・メソッドでしばしば登場する。

In [ ]:
```
def foo(x, y, z):       #z を印字して、x + y の結果を返すデモ関数
    print(z)
    return x + y

# 以下 2 つは同じ意味
print(foo(3, 4, "foo"))
print(foo(z = "foo", x = 3, y = 4))
```

変数と引数

関数・メソッドの呼び出しでは引数として与える変数に注意する必要がある。
少し面倒になるが、具体例を示しながら説明する:

In [ ]:
```
def foo(lst):             # リスト(配列)に要素 99 を追加する関数
    lst.append(99)

a = [1, 2, 3]                    # リストをつくる
tmp = a                          # tmp に a を代入、a を保存したつもり
```

In [ ]:
```
tmp                              # 中身を確認
```

In [ ]:
```
foo(a)                           # 関数を呼ぶ
```

In [ ]:
```
tmp                              # tmp を確認、あれ？
```


Python による代入・引数の引き渡しでは、変数・オブジェクトの複製はおこなわれず 変数・オブジェクトへの参照(情報)が渡されるだけである。 したがって、最初のセル 5 行目の = による代入では、右辺aの参照のみをtmpに代入している。 続く関数呼び出しでは、 a の参照を関数のローカル変数 some_list にコピーしている。 したがって、 a, tmp, some_list はすべて同じオブジェクトを参照していることから、 関数内で、 some_list に対しておこなった変更が、 tmp に及ぶ。

リストの例

Python では変数・オブジェクトが巨大となっても、代入・関数への引き渡しのコストが抑えられている。 一方で、変数・オブジェクトの内容のコピーを必要とするときは明示的に指定する必要がある。

In [ ]:
```
a = [1,2,3]      # リスト（配列を作る）
b = a            # b に a を代入（a への参照を作る）
print(b)
a.append(4)      # 配列に加えてみる
print(b)         # あれ？
```

In [ ]:
```
a = [1,2,3]      # リスト（配列を作る）
b = a.copy()     # コピーしたリストを代入する
print(b)
a.append(4)      # 配列に加えてみる
print(b)         # OK
```

Python の型

Python では型付けは動的に行われ、変数宣言は不要である。 以下の例で type() は変数・オブジェクトの型を返す関数:

In [ ]:
```
a = "foo"
print(type(a))

a = 5
print(type(a))

a = a / 4
print(type(a))
```

もちろん、あきらに矛盾する場合はエラーとなる:

In [ ]:
```
a = "foo" + 5
```

型・クラス検査

変数・オブジェクトが特定の型・クラスかどうかを検査するには、 isinstance()を使う。 以下は動的な型付けの例:

In [ ]:
```
a = "foo"
print(isinstance(a, int))
print(isinstance(a, str))

a = 5
print(isinstance(a, int))
print(isinstance(a, str))

a = a / 4
print(isinstance(a, float))
```

オブジェクト属性とメソッド

オブジェクトは属性（Attribute）とメソッドを持つ。 属性はオブジェクト内部に置かれる他の変数・オブジェクトであり、 メソッドはオブジェクトに結びつく関数で、オブジェクト内部のデータアクセスに使われる。 属性、メソッドいずれも`obj.attribute_name` によってアクセスされる。 jupyter では変数・オブジェクトの直後にカーソルのある状態で Tab キーを押せば、属性一覧を得ることができる:

In [ ]:
```
# 最終行末尾ににカーソルを合わせて Tab キーを押す：
a="foo"
a.
```

属性・メソッドを得るには、`getattr()`を使う:

In [ ]:
```
a="foo"                          # a は文字列型
a_count = getattr(a,'count')     # a.count() 関数を a_count という変数に代入
a_count("o")                     # Count the number of "o" in "foo"
```

モジュールの読み込み(import)

Python では多様な拡張機能がモジュールとして提供されており、 これらは `import` によって読み込むことができる。 演習率 `\pi` や、指数関数`exp()`は数学関数モジュール `math` に定義されており、 モジュール読み込み、利用は、 `import`, `from`, `as`を利用して以下のようにおこなえる:

In [ ]:
```
# 基本的な方法
import math
print(math.pi)
print(math.exp(1.0))
print()

# モジュールの一部のみ読み込む
from math import pi, exp
print(pi)
print(exp(1))
print()

# モジュール名が長いので別名で読み込む
import math as m
print(m.pi)
print(m.exp(1))
```

Python プログラムであればモジュールとして読み込むことができる。 また、複数のファイルでモジュールを構築する場合はそれらをまとめてディレクトリに配置、パッケージ化することもできる。この場合、ディレクトリに `__init__.py` ファイルを置きパッケージディレクトリであることを示す。

