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。

Python の組み込み型(再掲)

前述したとおり、Python には以下のような組み込み型がある:

スカラー型:

1. 数
2. ブーリアン（真理値判定）
3. 文字列
4. None

データ構造:

1. リスト
2. タプル
3. 集合
4. 辞書

ここではリスト型から説明する:

リスト型

リストはオブジェクトの配列(sequence)で、リストの長さおよび要素は変更可能（mutable）。
リストは [ ] で囲い、コンマ区切りで要素を記述する。
リスト要素の型に制約はなく、同じリストに異なる型を混在させることもできる:

In [3]:
```
list_a = ["Apple", 1, 3/4]                          # 文字列、整数、浮動小数点が混在したリスト
print("1.", list_a)
print("2.", list_a[0], list_a[1], list_a[2])        # 文字列型のような添字
print("3.", list_a[-1], list_a[1:2])                # 負の添字、スライスも使えます

list_b = []                                         # 空のリスト、要素はこの後の処理で追加される（おそらく）
print("4.", list_b)

list_c = list("abcd")                               # 文字列をリストに型変換（既出）
print("5.", list_c)
```

```
1. ['Apple', 1, 0.75]
2. Apple 1 0.75
3. 0.75 [1]
4. []
5. ['a', 'b', 'c', 'd']
```

リストもオブジェクトですので、リストの要素とすることもできます。
つまりリストの入れ子（ネスト）も扱えます。下記の例は 2 次元の 3x3 のリストです:

In [99]:
```
[[0, 1 , 2], [3, 4 , 5], [6, 7, 8]]
```

Out[99]:
```
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

リストの要素を変更する組み込み関数・メソッドとして以下がある:

1. 長さを知るには `len(リスト)` （下記 1.）
2. 要素を追加するには `list.append(要素)` （下記 2.）
3. 要素を指定した位置に追加するには `list.insert(インデックス,要素)` （下記 3.）
4. 要素を削除するには `list.remove(要素)` 要素が複数ある場合はリスト先頭側が削除される（下記 4.）
5. 要素を指定した位置から削除するには `list.pop(インデックス)` 戻り値は削除された要素（下記 5.）

In [6]:
```
foo_bar1 = ["foo", "bar", "baz", "qux", "quux"] # "corge", "graul
t", "garply","waldo", "fred", "plugh", "xyzzy", "thud" とつづく
print("1.", len(foo_bar1))

foo_bar1.append("corge")
print("2.", foo_bar1)

foo_bar1.insert(3, "foo")                          # 3 番目に "foo"
を挿入
print("3.", foo_bar1)

foo_bar1.remove("foo")                             # "foo" を削除、先
頭の "foo" だけが削除される
print("4.", foo_bar1)
print("5.", foo_bar1.pop(2))                       # "2" 番目を削除、
要素を出力
print("6.", foo_bar1)
```

```
1. 5
2. ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
3. ['foo', 'bar', 'baz', 'foo', 'qux', 'quux', 'corge']
4. ['bar', 'baz', 'foo', 'qux', 'quux', 'corge']
5. foo
6. ['bar', 'baz', 'qux', 'quux', 'corge']
```

リスト中の要素の有無を検査するには `in` または `not in` 演算子を使う。
in演算子の左辺の要素があれば `True` を、その他では `False` を返し、 `not in` では逆を返す。(下記 1-4)

要素の位置まで知るには `list.index(要素)` を、要素の数は`list.count(要素)` を使う。(下記 5-6)
いずれも該当する要素がなければ、エラーとなるので注意:

In [15]:
```
foo_bar2 = ["foo", "bar", "baz"]
print("1.", "foo" in foo_bar2)
print("2.", "baz" in foo_bar2)

print("3.", "qux" in foo_bar2)
print("4.", "qux" not in foo_bar2)

print("5.", foo_bar2.index("baz"))
print("6.", foo_bar2.count("bar"))
```

```
1. True
2. True
3. False
4. True
5. 2
6. 1
```


リストは、文字列同様に+演算子で結合させることができる。 リスト変数がすでに定義されているときは、そのリストを拡張するlist.extend(リスト)も使える。

付け加えると、上の例で、A. B. とも同じリストを返しますが、A. の方がより多くの資源を必要とするため、B. の方が好ましい。

In [105]:
```
print(["foo", "bar"]+["baz", "qux"])
foo_bar = ["foo", "bar"]

foo_bar2 = foo_bar + ["baz", "qux"]        # A. あらたにリストが作られます
print(foo_bar2)

foo_bar.extend(["baz", "qux"])             # B. 既存のリストを拡張します、Python ではこちらが推奨される
print(foo_bar)
```

```
['foo', 'bar', 'baz', 'qux']
['foo', 'bar', 'baz', 'qux']
['foo', 'bar', 'baz', 'qux']
```

逆に代入= では左辺に変数を,区切りで並べ、リスト要素を分解できる。(下記 1.,5.)
左辺の変数列に *_ があれば未分解の要素はこれに代入される。(下記 2-4)
おなじ記法で変数の値も交換できる（プログラム言語によっては一時変数が必要)(下記 6, 7):

In [18]:
```
list_a = ["Apple", 1, 3/4, "青森"]
fruite, number, weight, origin = list_a                    #リスト要素を分解して代入、要素数は合わせること
print("1.", fruite, number, weight, origin)

fruite_a, *_ = list_a                                      # *_ を使えば要素数を合っていなくても良い
print("2.", fruite_a)
*_, number_b, weight_b, origin_b = list_a                  # *_ は冒頭、途中でも可能
print("3.", number_b, weight_b, origin_b)
fruite_c, *_, origin_c = list_a
print("4.", fruite_c, origin_c)


list_o = ["Orange", 2, [3/4, 5/8], "和歌山"]
fruite_d, number_d, [weight1, weight2], origin_d = list_o #入れ子の場合、リストの構造とあわせること
print("5.", fruite_d, number_d, weight1, weight2, origin_d)

a = 1
b = 0
print("6.", a, b)
a,b = b,a                                                  # a, b を交換する、C では tmp = a; a = b ; b = tmp; となる
print("7.", a, b)
```

```
1. Apple 1 0.75 青森
2. Apple
3. 1 0.75 青森
4. Apple 青森
5. Orange 2 0.75 0.625 和歌山
6. 1 0
7. 0 1
```

リストを並べ替えるメソッドとして以下のようなものがある:

- リストの順番を逆にするには `list.reverse()`(下記 1.)
- リストを昇順にソートするには `list.sort()`
  `list.sort(key=ソートキー)`でソートキー（大小を判断する基準）を指定することもできる(下記 2-3)
  これらは、インプレースなメソッドであり元のリストは書き換えられる。

元のリストを残したい場合は `sorted(リスト)` でリストを作る(下記 4):

In [22]:
```
numbers_a = [ 7, 0, 6, 1, 5, 2, 4, 3]
numbers_a.reverse()
print("1.", numbers_a)

strings_b = ["The", "quick", "brown", "fox", "jumps"]
strings_b.sort()
print("2.", strings_b)                    # A-Za-z の順番
strings_b.sort(key=len)                     # 単語の短い順
print("3.", strings_b)

numbers_c = [ 7, 0, 6, 1, 5, 2, 4, 3]
numbers_d = sorted(numbers_c)
print("4.", numbers_c, numbers_d)
```

```
1. [3, 4, 2, 5, 1, 6, 0, 7]
2. ['The', 'brown', 'fox', 'jumps', 'quick']
3. ['The', 'fox', 'brown', 'jumps', 'quick']
4. [7, 0, 6, 1, 5, 2, 4, 3] [0, 1, 2, 3, 4, 5, 6, 7]
```

リスト内包表記

Python では内包表記(complehensive)が利用できる。内包表記も Python の特徴の一つである。
平方のリストは以下のように得ることができる。:

In [26]:
```
squares1 = []
for x in range(6):
    squares1.append(x**2)
squares1
```

Out[26]: `[0, 1, 4, 9, 16, 25]`

squares として `[0, 1, 4, 9, 16, 25]` が得られる。
これを内包表記を用いて書き換えると、以下のように一行で書けプログラムが読みやすくなる:

In [25]:
```
squares2 = [x**2 for x in range(6)]
squares2
```

Out[25]: `[0, 1, 4, 9, 16, 25]`

もちろん内包表記をネスト（入れ子）にすることも可能である:

In [ ]:
```
table =  [[x*y for y in range(3)] for x in range(3)]
vector = [x*y for y in range(3) for x in range(3)]
```

内包表記はforに加えてifを使うこともできる:

In [29]:
```
words = ["cat", "dog", "elephant", None, "giraffe"]
lengths = [len(w) for w in words if w != None]
lengths
```

Out[29]: `[3, 3, 8, 7]`

if を使わないと、エラーになる:

In [30]:
```
words = ["cat", "dog", "elephant", None, "giraffe"]
lengths2 = [len(w) for w in words]
lengths2
```

```
---------------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-30-defd7322a98c> in <module>()
      1 words = ["cat", "dog", "elephant", None, "giraffe"]
----> 2 lengths2 = [len(w) for w in words]
      3 lengths2

<ipython-input-30-defd7322a98c> in <listcomp>(.0)
      1 words = ["cat", "dog", "elephant", None, "giraffe"]
----> 2 lengths2 = [len(w) for w in words]
      3 lengths2

TypeError: object of type 'NoneType' has no len()
```

タプル(tuple)型

タプルはオブジェクトの配列(sequence)だが、リストと異なり変更することはできない(immutable)。タプルは `()` で囲い、コンマ区切りで要素を記述する。要素の変更を伴わない機能はリストとほぼ同じ:

In [31]:
```
tuple_a = ("Apple", 1, 3/4)                    # 文字列、整数、浮動小数点が混在したタプル
print("1.", tuple_a)
print("2.", tuple_a[0], tuple_a[1], tuple_a[2])      # 文字列型のような添字

print("3.", len(tuple_a))

print("4.", "Apple" in tuple_a)
print("5.", tuple_a.index(3/4))
print("6.", tuple_a.count("Apple"))

fruite, number, weight = tuple_a
print("7.", fruite, number, weight)
```

```
1. ('Apple', 1, 0.75)
2. Apple 1 0.75
3. 3
4. True
5. 2
6. 1
7. Apple 1 0.75
```

タプルの要素を変更するには、`list(タプル)`でリストを生成し、 要素を更新後`tuple(リスト)`に戻せばよい:

In [32]:
```
tuple_b = ("Apple", 1, 3/4)
tuple_b[0] = "Orange"                    # エラーになる。
```

```
---------------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-32-79657b5a96bc> in <module>()
      1 tuple_a = ("Apple", 1, 3/4)
----> 2 tuple_a[0] = "Orange"                 # エラーになる。

TypeError: 'tuple' object does not support item assignment
```

In [35]:
```
tuple_c = ("Apple", 1, 3/4)
print("1.", tuple_c)
list_c = list(tuple_c)
list_c[0] = "Orange"

print("2.", list_c)
tuple_c = tuple(list_c)
print("3.", tuple_c)
```

```
1. ('Apple', 1, 0.75)
2. ['Orange', 1, 0.75]
3. ('Orange', 1, 0.75)
```

generator 式(参考)

Python ではタプル、()、で内包表記を利用することはできないが、 小括弧内()に内包表記（厳密には違う）を記述するとタプルではなく generator オブジェクトがつくられる。

したがって、以下のプログラムは動作するがオブジェクト squares_gen はタプルではない。

```
Python:generator_sample.py
squares_gen = (x**2 for x in range(6))
for x in squares_gen:
    print(x)
```

辞書(dict)型

辞書型は重要な Python 組み込みデータ構造である。
一般的にはハッシュあるいは連想配列とも呼ばれる。
辞書は可変長でキーおよび値(key-value)ペアから構成されている。
辞書は:でキー:値を、{}で囲うことで生成することができる。
キーを指定した辞書要素へのアクセスは辞書[キー]でおこなう。
削除は del 辞書[キー] 式あるいは 辞書.pop(キー)メソッドでおこなえる:

In [37]:
```
dict_a = {"apple":100, "orange":200}    #

print("1,", dict_a["apple"])
dict_a["apple"] = 0                     # 辞書の変更
print("2.", dict_a["apple"])

print("3.", dict_a)
dict_a["peach"] = 200                   # 要素の追加
print("4.", dict_a)
del dict_a["apple"]                     # 要素の削除
print("5.", dict_a)

number = dict_a.pop("peach")            # 要素を削除し、値を返す
print("6.", dict_a)
print("7.", number)

dict_b = {}                             # 空の辞書も作れる
print("8.", dict_b)
```

```
1, 100
2. 0
3. {'apple': 0, 'orange': 200}
4. {'apple': 0, 'orange': 200, 'peach': 200}
5. {'orange': 200, 'peach': 200}
6. {'orange': 200}
7. 200
8. {}
```

辞書のキーの有無を検査するには in 演算子が使える:

In [38]:
```
dict_a = {"apple":100, "orange":200}    #

print("1.", "apple" in dict_a)
print("2.", "kiwi" in dict_a)
```

```
1. True
2. False
```

辞書をまとめて更新するには、辞書.update(辞書)を使う:

In [39]:
```
dict_a = {"apple":100, "orange":200}
print("1.", dict_a)
dict_a.update({"kewi":300, "peach":50})
print("2.", dict_a)
dict_a.update({"apple":0, "orange":0})
print("3.", dict_a)                             # 既存のキーがあれば値が更新される
```

```
1. {'apple': 100, 'orange': 200}
2. {'apple': 100, 'orange': 200, 'kewi': 300, 'peach': 50}
3. {'apple': 0, 'orange': 0, 'kewi': 300, 'peach': 50}
```

キーと値が別のリストで与えられるときは、zip(リスト1, リスト2)が便利である。

In [40]:
```
fruite_names = ["apple", "orange"]
fruite_numbers = [100,200]
dict_a = {}

for key, value in zip(fruite_names, fruite_numbers):  # 普通はこう書く
    dict_a[key] = value

print("1.", dict_a)

dict_b = dict(zip(fruite_names, fruite_numbers))      # こういう書き方もできる
print("2.", dict_b)
```

```
1. {'apple': 100, 'orange': 200}
2. {'apple': 100, 'orange': 200}
```

キーに対応する値を返す 辞書.get(キー)は、キーが存在しなければ Noneを返すが、 第二引数に既定値を与え 辞書.get(キー,既定値)とすることもできる:

In [50]:
```
dict_a = {"apple":100, "orange":200}
print("1.", dict_a.get("kiwi"))              # print None

print("2.", dict_a.get("kiwi", 0))           # return default value 0 instead of None
```

```
None
0
```

辞書のキーの制約

辞書型のキーは immutable オブジェクトしかとれないことに注意すること。
つまり数型、文字列型、タプルなどはエラーとならないが、リストは mutable オブジェクトでありキーとすることはできない。 以下の最終行はエラーとなる:

In [41]:
```
dict_a = {"apple":[100,200]}      # No error
print("1.", dict_a)

dict_a = {100, "apple"}           # No error
print("2.", dict_a)

dict_a = {(100,200), "apple"}     # No error
print("3.", dict_a)

dict_a = {[100, 200], "apple"}    # エラーになります
```

```
1. {'apple': [100, 200]}
2. {'apple', 100}
3. {'apple', (100, 200)}
---------------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-41-078dc526b83b> in <module>()
      8 print("3.", dict_a)
      9
---> 10 dict_a = {[100, 200], "apple"}    # エラーになります

TypeError: unhashable type: 'list'
```

辞書型の内包表記

内包表記は辞書型でも使うことができる:

In [42]:
```
words = ["cat", "dog", "elephant", None, "giraffe"]
length_dir = {w:len(w) for w in words if w != None}       # 内包表
記
length_dir
```

Out[42]:
```
{'cat': 3, 'dog': 3, 'elephant': 8, 'giraffe': 7}
```

集合(set)型

集合型重複のない(ユニークな)要素からなる配列である。
集合は `{, }` で要素を囲うか、`set()`で生成される。 ただし、空の集合は `{}` では生成されない（代わりに辞書が生成される）:

In [45]:
```
set_a = {"apple", "orange", "kiwi", "apple"} # apple が重複した
print("1.", set_a)                                  # apple は重複
していない

list_b = list(set([1,2,3,4,4,3,2,1]))        # いい加減に作ったリス
トから重複を除きたい
print("2.", list_b)                          # リスト -> 集合 -> リ
スト、とする
```

```
1. {'orange', 'kiwi', 'apple'}
2. [1, 2, 3, 4]
```

2 つの集合の和集合 $\bigcup$ は 集合.union(集合) または演算子 |、
共通部分 $\bigcap$ は 集合.intersection(集合) または演算子 &　から得られる:

In [50]:
```
set_b = {1,2,3,4,5,6}
set_c = {4,5,6,7,8,9}

print("1.", set_b.union(set_c))
print("2.", set_b | set_c)
print("3.", set_b.intersection(set_c))
print("4.", set_b & set_c)
```

```
1. {1, 2, 3, 4, 5, 6, 7, 8, 9}
2. {1, 2, 3, 4, 5, 6, 7, 8, 9}
3. {4, 5, 6}
4. {4, 5, 6}
```

集合の内包表記

内包表記は集合型、{}、でも使うことができる: 集合型なので、リストと異なり重複する要素は除かれる。

In [73]:
```
words = ["cat", "dog", "elephant", None, "giraffe"]
length_set = {len(w) for w in words if w != None}
print(length_set)
```

```
{8, 3, 7}
```

