Python 3

По мотивам Python Tips, Tricks, and Hacks

- Четыре вида кавычек

```
print ("""I wish that I'd never heard him say, "'She said, "He said, 'Give me five dollars'"'"""")
```

- Проверка на True

```
my_object = 'Test' # True example
# my_object = '' # False example

if len(my_object) > 0:
    print ('my_object is not empty')

if len(my_object): # 0 will evaluate to False
    print ('my_object is not empty')

if my_object != '':
    print ('my_object is not empty')

if my_object: # an empty string will evaluate to False
    print ('my_object is not empty' )
```

- Проверка подстроки

```
string = 'Hi there' # True example
# string = 'Good bye' # False example
if string.find('Hi') != -1:
    print ('Success!' )
```

```
string = 'Hi there' # True example
# string = 'Good bye' # False example
if 'Hi' in string:
    print ('Success!' )
```

- # Деление в Python 2 и Python 3

```
5/2 # Returns 2
5.0/2 # Returns 2.5

(5)/2 # Returns 2.5
5//2 # Returns 2
```

```
5/2 # Returns 2.5
5.0/2 # Returns 2.5

(5)/2 # Returns 2.5
5//2 # Returns 2
```

- # Lambda функции

```
def add(a,b): return a+b

add2 = lambda a,b: a+b

squares = map(lambda a: a*a, [1,2,3,4,5])
# squares is now [1,4,9,16,25]]
#In real <map object at 0x7fa230383080>
```

- # Iterator Type and Generators

```
container.__iter__()
#Return an iterator object

iterator.__iter__()
#Return the iterator object itself

iterator.__next__()
#Return the next item from the container.
If there are no further items, raise the StopIteration exception.
```

Generator functions use yield expression

```
def city_generator():
    yield("London")
    yield("Hamburg")
    yield("Konstanz")
    yield("Amsterdam")
    yield("Berlin")
    yield("Zurich")
    yield("Schaffhausen")
    yield("Stuttgart")
```

Списки, Comprehensions, операции map и filter

```
numbers = [1,2,3,4,5]
squares = []
for number in numbers:
    squares.append(number*number)
# Now, squares should have [1,4,9,16,25]
```

```
numbers = [1,2,3,4,5]
numbers_under_4 = []
for number in numbers:
    if number < 4:
        numbers_under_4.append(number)
# Now, numbers_under_4 contains [1,4,9]
```

```
numbers = [1,2,3,4,5]
squares = [number*number for number in numbers]
# Now, squares should have [1,4,9,16,25]
```

```
numbers = [1,2,3,4,5]
numbers_under_4 = [number for number in numbers if number < 4]
# Now, numbers_under_4 contains [1,2,3]
```

```
numbers = [1,2,3,4,5]
numbers_under_4 = filter(lambda x: x < 4, numbers)
# Now, numbers_under_4 contains [1,2,3]
```

```
numbers = [1,2,3,4,5]
squares = map(lambda x: x*x, numbers)
# Now, squares should have [1,4,9,16,25]
```

Выражения-генераторы

```
numbers = (1,2,3,4,5) # Since we're going for efficiency, I'm using a tuple instead of a list ;)
squares_under_10 = (number*number for number in numbers if number*number < 10)
# squares_under_10 is now a generator object, from which each successive value can be gotten by calling .next()

for square in squares_under_10:
    print(square)
# prints '1 4 9'


for x in (0,1,2,3):
    for y in (0,1,2,3):
        if x < y:
            print ((x, y, x*y),)

# prints (0, 1, 0) (0, 2, 0) (0, 3, 0) (1, 2, 2) (1, 3, 3) (2, 3, 6)


print([(x, y, x * y) for x in (0,1,2,3) for y in (0,1,2,3) if x < y])
# prints [(0, 1, 0), (0, 2, 0), (0, 3, 0), (1, 2, 2), (1, 3, 3), (2, 3, 6)]
```

Словари и Dict Comprehensions

```
dict(a=1, b=2, c=3)
# returns {'a': 1, 'b': 2, 'c': 3}

dictionary = {'a': 1, 'b': 2, 'c': 3}
dict_as_list = dictionary.items()
#dict_as_list now contains [('a', 1), ('b', 2), ('c', 3)]


dict_as_list = [['a', 1], ['b', 2], ['c', 3]]
dictionary = dict(dict_as_list)
# dictionary now contains {'a': 1, 'b': 2, 'c': 3}


dict_as_list = [['a', 1], ['b', 2], ['c', 3]]
dictionary = dict(dict_as_list, d=4, e=5)
# dictionary now contains {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

emails = {'Dick': 'bob@example.com', 'Jane': 'jane@example.com', 'Stou': 'stou@example.net'}
email_at_dotcom = dict( [name, '.com' in email] for name, email in emails.iteritems() )
# email_at_dotcom now is {'Dick': True, 'Jane': True, 'Stou': False}
email_at_dotcom = { name: '.com' in email for name, email in emails.iteritems() }
```

Выбор значений в выражении

```
test = True
# test = False
result = 'Test is True' if test else 'Test is False'
# result is now 'Test is True'

test1 = False
test2 = True
result = 'Test1 is True' if test1 else 'Test1 is False, test2 is True' if test2 else 'Test1 and Test2 are both False'

test = True
# test = False
result = test and 'Test is True' or 'Test is False'
# result is now 'Test is True'

test = True
# test = False
result = ['Test is False','Test is True'][test]
# result is now 'Test is True'
```

Функции и значения по умолчанию

```
def function(item, stuff = []):
    stuff.append(item)
    print(stuff)

function(1)
# prints '[1]'

function(2)
# prints '[1,2]' !!!
```

```
def function(item, stuff = None):
    if stuff is None:
        stuff = []
    stuff.append(item)
    print(stuff)

function(1)
# prints '[1]'

function(2)
# prints '[2]', as expected
```

Функции с переменным числом параметров

```
def do_something(a, b, c, *args):
    print(a, b, c, args)

do_something(1,2,3,4,5,6,7,8,9)
# prints '1, 2, 3, (4, 5, 6, 7, 8, 9)'

def do_something_else(a, b, c, *args, **kwargs):
    print(a, b, c, args, kwargs)

do_something_else(1,2,3,4,5,6,7,8,9, timeout=1.5)
# prints '1, 2, 3, (4, 5, 6, 7, 8, 9), {"timeout": 1.5}'

args = [5,2]
pow(*args)
# returns pow(5,2), meaning 5^2 which is 25

def do_something(actually_do_something=True, print_a_bunch_of_numbers=False):
    if actually_do_something:
        print('Something has been done‘)
        #
        if print_a_bunch_of_numbers:
            print(range(10))

kwargs = {'actually_do_something': True, 'print_a_bunch_of_numbers': True}
do_something(**kwargs)

# prints 'Something has been done', then '[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]'
```

Set и Frozen set

Неупорядоченный набор уникальных значений, так же как словарь представляет собой неупорядоченный набор пар ключ / значение.

Функции преобразования

`set(x)`	создает множество, содержащее элементы коллекции x
`frozenset(x)`	создает неизменное множество, содержащее элементы коллекции x

```
>>> Set((1, 2, 3, 4)) # case 1
>>> Set(1, 2, 3, 4) # case 2
```

инициализатор принимает единственный итерируемый аргумент

```
set(x**2 for x in S)
```

создает множество, содержащее квадраты всех элементов из S

Set и Frozen set

```
for x in S:
```

представляет собой проход по элементам S в произвольном порядке

```
if "a" in S:
    ...
```

проверяет вхождение во множество

Оператор

```
| объединение
& пересечение
^ симметрическая разность
- асимметрическая разность
==  != проверки равенства и неравенства
<  <=  >=  > проверки на подмножество и супермножество
```

Set и Frozen set

Методы
:

`len(S)`	возвращает количество элементов во множестве
`S.add(x)`	добавляет x во множество
`S.update(s)`	добавляет все элементы последовательности s во множество
`S.remove(x)`	удаляет x из множества (если x не существует во множестве, то вызывается LookupError)
`S.discard(x)`	удаляет x из множества (если x не существует во множестве – ничего не делает)
`S.pop()`	выталкивает произвольный элемент (вызывает LookupError если элемент не существует)
`S.clear()`	удаляет все элементы из множества
`S.copy()`	создает новое множество
`s.issuperset(S)`	проверяет, является ли S супермножеством s
`s.issubset(S)`	проверяет, является ли S подмножеством s
`s.isdisjoint(S)`	возвращает True, если множество s не имеет общих элементов с S

View object

Идея – изменить методы .keys(), .values() and .items() так, чтобы они возвращали более легковесный объект, чем лист и, тем самым, избавиться от методов .iterkeys(), .itervalues() and .iteritems().

Python 2.x

```
for k, v in d.iteritems():
    ...
```

```
a = d.keys()
```

Python 3.x

```
for k, v in d.items():
    ...
```

```
a = list(d.keys())
```

View object

Два пути…

.keys(), .values() и .items() возвращают итератор, аналогично тому, как это делали .iterkeys(), .itervalues() и .iteritems() в Python 2.x

более хороший и действующий сейчас

методы возвращают объекты, подобные множеству ==set (для .keys() и .items()) или мультимножеству == bag (для .values()), которые не содержат копий keys, values или items, вместо этого ссылаются на исходный словарь и берут значения из словаря по мере необходимости

И можно продолжать испол[illegible]

```
a = d.items()
for k, v in a: ...
# And later, again:
for k, v in a: ...
```

Но теперь также можно сделать так

```
if a.keys() == b.keys(): ...
```

и аналогично для .items().

Возвращаемые объекты полностью совместимы с типами set и frozenset

```
set(d.keys()) == d.keys()
```

```
list(d.items()) == list(zip(d.keys(), d.values()))
```