Functional programming

Amit Kotlovski
Open-Source Consultancy
http://amitkot.com

Agenda

- Functions in depth
- Functional tools
- Manipulating functions

Functions in depth

- Default values
- Named args
- Keyword args
- Docstrings
- doctest
- lambda functions

Basic function declaration and usage

```
def power2(num):
    result = num**2
    return result
```

```
>>> power2(2)
4
>>> power2(11)
121
```

Keyword arguments

```
def weather(temperature, humidity, rainfall,
            wind, pressure, visibility, clouds):
    print 'what a fine day!'
```

```
# Bad :(
>>> weather(30, 0.55, 0, (17, 300), 1010, 10.0,
            ‘Few 1066m')
what a fine day!

# Good :)
>>> weather(temperature=30, humidity=0.55,
            rainfall=0, wind=(17, 300), pressure=1010,
            visibility=10.0, clouds='Few 1066m')
what a fine day!
```

Default arguments

```
def foo(name, number, color='blue'):
    print('color is ' + color)
```

```
>>> foo('Jay', 32, 'yellow')
color is yellow
>>> foo('Joe', 34)
color is blue
```

Default Arguments are always the last ones

```
# Good :)
def foo(name, number, color='blue'):
    print('color is ' + color)
```

```
# Bad :(
def foo(name, color='blue', number):
    print('color is ' + color)
```

Default value evaluated only once

```
# bad :(
def add_animal(animal, zoo=[]):
    zoo.append(animal)
    return zoo

>>> print(add_animal('Canary'))
['Canary']
>>> print(add_animal('Unicorn'))
['Canary', 'Unicorn']
>>> print(add_animal('Narwhal'))
['Canary', 'Unicorn', 'Narwhal']
```

Default value evaluated only once

```
# good :)
def add_animal(animal, zoo=None):
    if zoo is None:
        zoo = []
    zoo.append(animal)
    return zoo

>>> print(add_animal('Canary'))
['Canary']
>>> print(add_animal('Unicorn'))
['Unicorn']
>>> print(add_animal('Narwhal'))
['Narwhal']
```

Tuple packing and unpacking

```
>>> t1 = (1, 2, 3)
>>> a, b, c = t1      # tuple unpacking
>>> a
1
>>> b
2
>>> c
3

>>> t2 = a, b, c      # tuple packing
>>> t2
(1, 2, 3)
```

Arbitrary Argument Lists (*args)

```
def flexible(name, number, *args):
    print('number of unplanned arguments is',
          len(args))
    for arg in args:
        print('extra arg is ', arg)
```

```
>>> flexible('Dan', 17, 'loves apples',
             'loves skydiving')
number of unplanned arguments is 2
extra arg is loves apples
extra arg is loves skydiving
```

Arbitrary *keyword* argument lists (**kwargs)

```
def flexible(name, number, **kwargs):
    print(‘number of unplanned arguments is’,
          len(kwargs))
    for key in kwargs:
        print(’extra arg key:', key, ', value:',
            kwargs[key])
```

```
>>> flexible('Dan', 17, fruit='loves apples',
                sport='loves skydiving')
number of unplanned arguments is 2
extra arg key: fruit , value: loves apples
extra arg key: sport , value: loves skydiving
```

Unpacking Argument Lists

```
def greet(name, greeting):
    return "Hi {}, {}!".format(name, greeting)
```

```
>>> args = ["David", "nice to meet you"]
>>> greet(*args)
'Hi David, nice to meet you!'

>>> args = {"name": "David",
            "greeting": "nice to meet you"}
>>> greet(**args)
'Hi David, nice to meet you!'
```

docstring

```
def is_comment_line(line):
    """Checks if provided line is a comment.

    The function checks if the provided line is a comment
    by inspecting it's characters to see wether it starts
    with a a "#" char.

    :param line: the line to be checked
    :return: true if the line is indeed a comment, false otherwise
    """
    return line.startswith("#")
```

doctest

```
def is_comment_line(line):
    """Checks if provided line is a comment.

    The function checks if the provided line is a comment
    by inspecting it's characters to see wether it starts
    with a a "#" char.

    Example:
    >>> is_comment_line("what a nice line")
    False
    >>> is_comment_line("# this is a comment")
    True
    """
    return line.lstrip().startswith("#")

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

```
$ python -m doctest -v doctest_example.py
Trying:
    is_comment_line("what a nice line")
Expecting:
    False
ok
Trying:
    is_comment_line("# this is a comment")
Expecting:
    True
ok
1 items had no tests:
    doctest_example
1 items passed all tests:
   2 tests in doctest_example.is_comment_line
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

lambda functions

Defining anonymous functions

```
def last_digit(num):
    return num % 10
```

```
>>> sorted([106, 22, 8], key=last_digit)
[22, 106, 8]

>>> sorted([106, 22, 8], key=lambda x: x % 10)
[22, 106, 8]
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Lab

2a - Functions in depth

Functional tools

- map
- filter
- reduce
- list sorting
- operator module

Functions as objects

- Python functions are objects (first class citizens)
- Can be passed as parameters to other functions
- Can be returned as return values from functions

Functions can be assigned

Same as lists, tuples, etc.

```
def plus(x, y):
    return x + y
```

```
>>> plus_func = plus
>>> print(plus(23, 7))
30
>>> print(plus_func(23, 7))
30

>>> print("plus id is\t", id(plus))
plus id is        4578510912
>>> print("plus_func id is\t", id(plus_func))
plus_func id is   4578510912
```

Higher-order functions

Accept functions as arguments

```
def do(action, a, b):
    return action(a, b)

def plus(x, y):
    return x + y

c = do(plus, 5, 3)
print(c)
```

map

Runs a function on each item in the sequence

```
def times6(x):
    return x*6

>>> numbers = [1, 2, 3, 4]

>>> list(map(times6, numbers))
[6, 12, 18, 24]

>>> list(map(lambda x: x**2, numbers))
[1, 4, 9, 16]
```

Multiple sequences

```
map(func, p, q, …) -> func(p1, q1, …),
                      func(p2, q2, …),
                      ...
```

```
>>> list(map(lambda x, y, z: (x + y) * z,
             [1, 2, 3, 4],
             [5, 6, 7, 8],
             [9, 10, 11, 12]))

[54, 80, 110, 144]
```

Python 3: map returns iterable

(in Python 2 use itertools.imap)

```
# Python 3
>>> map(lambda x: x**2, [1, 2, 3, 4])
<map at 0x107360a90>

>>> list(map(lambda x: x**2, [1, 2, 3, 4]))
[1, 4, 9, 16]
```

```
# Python 2
>>> itertools.imap(lambda x: x**2, [1, 2, 3, 4])
<itertools.imap object at 0x10248ec10>

>>> list(itertools.imap(lambda x: x**2, [1, 2, 3, 4]))
[1, 4, 9, 16]
```

filter

Filter a sequence for items matching the condition

```
>>> l = [-2, -1, 0, 1, 2]

>>> def negative(x):
...     return x < 0
...
>>> list(filter(negative, l))
[-2, -1]

>>> list(filter(lambda x: x % 2 == 0, l))
[-2, 0, 2]
```

filtering True items

if `func` is **`None`**, return the items that are **`True`**

```
>>> list(filter(None, [0, '', [], 1, 2]))
[1, 2]
```

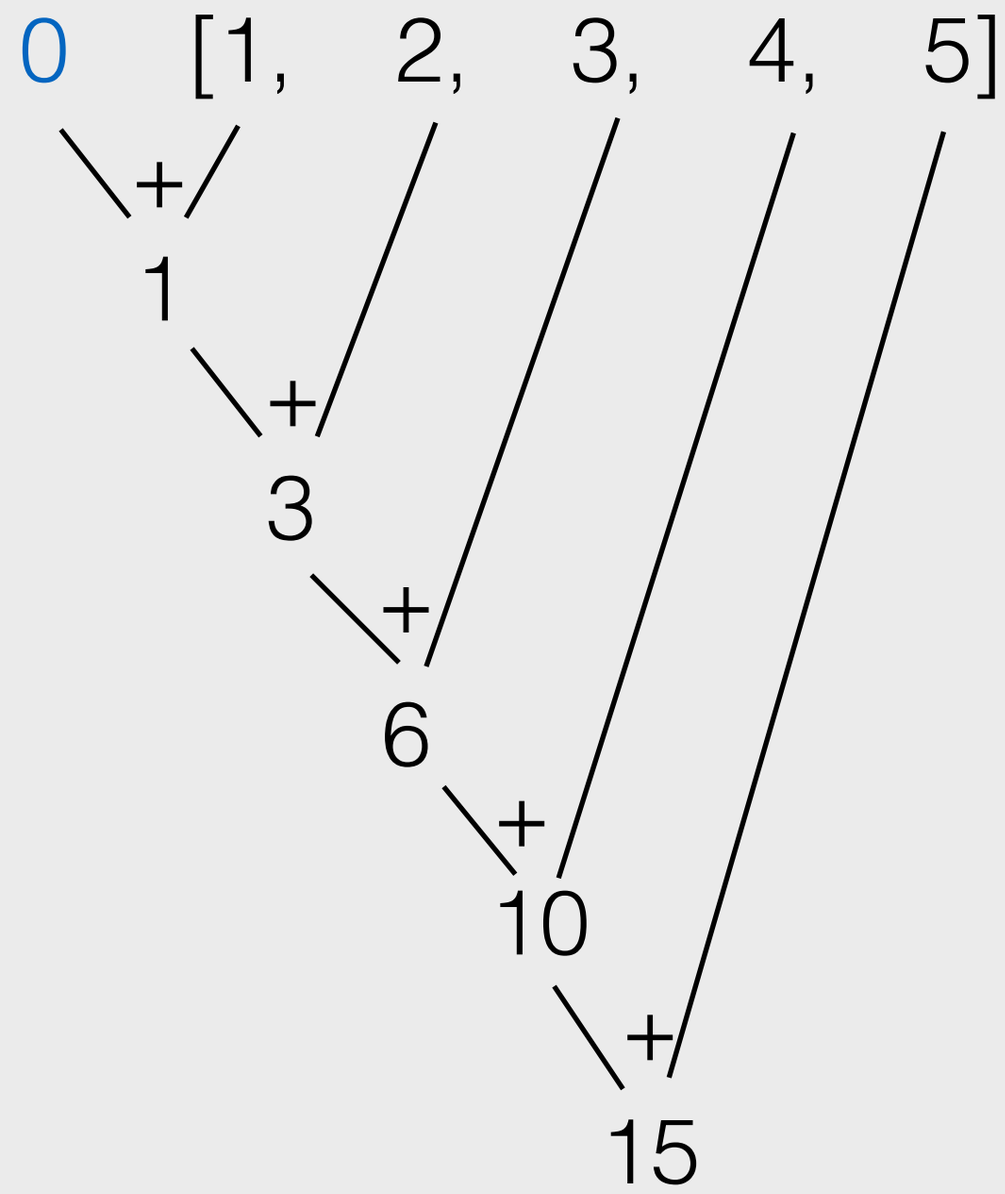
filtering data structures

```
>>> list(filter(str.islower, 'ABCabc'))
['a', 'b', 'c']

>>> list(filter(lambda x: x < 3, (1, 2, 3, 4, 5)))
(1, 2)

>>> list(filter(lambda x: x < 3,
                {1: 'a', 2: 'b', 3: 'c'}))
[1, 2]
```

reduce



functools.reduce()

Replace the first two values with the result of the function for them, then run again until the sequence is reduced to single value.
Optionally provide initial value for computation.

```
>>> functools.reduce(lambda x, y: x + y,
                     [1, 2, 3, 4, 5],
                     initial=0)
15
```

“So now reduce(). This is actually the one I've always hated most, because, apart from a few examples involving + or *, almost every time I see a reduce() call with a non-trivial function argument, I need to grab pen and paper to diagram what's actually being fed into that function before I understand what the reduce() is supposed to do.”

– Guido van van Rossum, blog post on March 10, 2005

functools.reduce()

Demoted in Python 3 from a builtin function in the global scope to the **functools** module.

list.sort() and sorted()

list.sort() sorts in place, sorted() returns a sorted list

```
>>> l = [11, 2, 333, 44, 5555]

>>> print(sorted(l))
[2, 11, 44, 333, 5555]
>>> l
[11, 2, 333, 44, 5555]

>>> print(l.sort())
None
>>> l
[2, 11, 44, 333, 5555]
```

using a comparator

```
>>> sorted(l, cmp=lambda a, b: len(str(a)) - len(str(b)))
[2, 11, 44, 333, 5555]
```

using a key function

The key function takes an item and returns the key for sorting it

```
>>> sorted([11, 2, 333, 44, 5555],
           key=lambda x: len(str(x)))
[2, 11, 44, 333, 5555]
```

Sorting in Python 3

Using key function is the default, to use comparator use functools.cmp_to_key(), e.g.:

```
def last_digit(a, b):
    return int(str(a)[-1]) - int(str(b)[-1])

>>> import functools
>>> l = [13, 24, 31, 42, 5555]
>>> sorted(l, key=functools.cmp_to_key(last_digit))
[31, 42, 13, 24, 5555]
```

`operator` module

- A collections of operators, implemented as functions

- Operators can replace lambdas in map, sort, etc.

using operators

```
>>> map(lambda a, b: a + b, [1, 2, 3], [11, 22, 33])
[12, 24, 36]

>>> map(operator.add, [1, 2, 3], [11, 22, 33])
[12, 24, 36]

>>> data = [(1, 2, 3), (11, 22, 33)]
>>> map(operator.itemgetter(1), data)
[2, 22]

>>> details = [dict(name='john', age=26),
               dict(name='george', age=32)]
>>> map(operator.itemgetter('name'), details)
['john', 'george']
```

more uses

```
class Person(object):
    def __init__(self, name, age):
            self.name = name
            self.age = age
    def __repr__(self):
            return '{{name={}, age={}}}'.format(self.name,
                                                self.age)
```

```
>>> data = [Person('john', 26), Person('george', 32)]

>>> map(operator.attrgetter('name'), data)
['john', ‘george']

>>> sorted(data, key=operator.attrgetter('age'))
[{name=john, age=26}, {name=george, age=32}]

>>> sorted(data, key=operator.attrgetter('name'))
[{name=george, age=32}, {name=john, age=26}]
```

Operation	Syntax	Function
Addition	a + b	add(a, b)
Attribute Access	obj.k	getattr(obj, k), also attrgetter(k)(ob)
Concatenation	seq1 + seq2	concat(seq1, seq2)
Containment Test	obj in seq	contains(seq, obj)
Division	a / b	div(a, b) (Py2 without __future__.division)
Division	a / b	truediv(a, b) (Py3, Py2 with__future__.division)
Division	a // b	floordiv(a, b)
Bitwise And	a & b	and_(a, b)
Bitwise Exclusive Or	a ^ b	xor(a, b)
Bitwise Inversion	~ a	invert(a)
Bitwise Or	a | b	or_(a, b)
Exponentiation	a ** b	pow(a, b)
Identity	a is b	is_(a, b)
Identity	a is not b	is_not(a, b)
Indexed Assignment	obj[k] = v	setitem(obj, k, v)
Indexed Deletion	del obj[k]	delitem(obj, k)
Indexing	obj[k]	getitem(obj, k), also itemgetter(k)(obj)
Left Shift	a << b	lshift(a, b)
Modulo	a % b	mod(a, b)
Multiplication	a * b	mul(a, b)
Negation (Arithmetic)	- a	neg(a)
Negation (Logical)	not a	not_(a)
Positive	+ a	pos(a)
Right Shift	a >> b	rshift(a, b)
Sequence Repetition	seq * i	repeat(seq, i)
Slice Assignment	seq[i:j] = values	setitem(seq, slice(i, j), values)
Slice Deletion	del seq[i:j]	delitem(seq, slice(i, j))
Slicing	seq[i:j]	getitem(seq, slice(i, j))
String Formatting	s % obj	mod(s, obj)
Subtraction	a - b	sub(a, b)
Truth Test	obj	truth(obj)
Ordering	a < b	lt(a, b)
Ordering	a <= b	le(a, b)
Equality	a == b	eq(a, b)
Difference	a != b	ne(a, b)
Ordering	a >= b	ge(a, b)
Ordering	a > b	gt(a, b)

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Lab

2b - Functional tools

Manipulating functions

- closures
- partial
- decorators

Nested function

Internal functions can be defined for internal use

```
def foo():
    def bar():
        return 42
    return bar() * 5

>>> print foo()
210
```

Closure

- When a function returns an internal function, and
- That function has access to local variables
- That are no longer accessible otherwise

```
def create_addX(x):
    def internal(y):
        return y + x
    return internal

>>> add7 = create_addX(7)
>>> print add7(23)
30

>>> add500 = create_addX(500)
>>> print add500(10)
510
```

```
def create_addX(x):
    def internal(y):
        return y + x
    return internal

>>> add500 = create_addX(500)
>>> print add500(10)
510

>>> add500.__closure__
(<cell at 0x112598be8: int object at 0x7ff581792780>,)

>>> len(add500.__closure__)
1

>>> type(add500.__closure__[0].cell_contents)
int

>>> print(add500.__closure__[0].cell_contents)
500
```

Implementing a closure that saves a function and it's first argument

```
def wrap(fn):
    def wrapper():
        return fn()
    return wrapper

def hello():
    return 'Hello!'
```

```
>>> wrapped_hello = wrap(hello)

>>> wrapped_hello()
'Hello!'
```

```
def wrap(fn, arg0):
    def wrapper():
        return fn(arg0)
    return wrapper

def greet(greeting):
    return f'{greeting}!'
```

```
>>> greet_hello = wrap(greet, 'Hello')

>>> greet_hello()
'Hello!'
```

```
def wrap(fn, arg0):
    def wrapper(arg1):
        return fn(arg0, arg1)
    return wrapper

def greet(greeting, name):
    return f'{greeting} {name}!'
```

```
>>> greet_hello = wrap(greet, 'Hello')

>>> greet_hello('Dan')
'Hello Dan!'
```

```
def wrap(fn, arg0):
    def wrapper(*args):
        return fn(arg0, *args)
    return wrapper

def greet(greeting, first, second):
    return f'{greeting} {first} and {second}!'
```

```
>>> greet_hello = wrap(greet, 'Hello')

>>> greet_hello('Dan', 'Anna')
'Hello Dan and Anna!'
```

```
def wrap(fn, arg0):
    def wrapper(*args, **kwargs):
        return fn(arg0, *args, **kwargs)
    return wrapper

def greet(greeting, first, second, upper=False):
    res = f'{greeting} {first} and {second}!'
    return res.upper() if upper else res
```

```
>>> greet_hello = wrap(greet, 'Hello')

>>> greet_hello('Dan', 'Anna')
'Hello Dan and Anna!'

>>> greet_hello('Dan', 'Anna', upper=True)
'HELLO DAN AND ANNA!'
```

Generalise to save multiple arguments

```
def partial(fn, *pargs):
    def wrapper(*args, **kwargs):
        new_args = pargs + args
        return fn(*new_args, **kwargs)
    return wrapper

def aa(a, b, c):
    print(a, b, c)
```

```
>>> a = partial(aa, 'nice')
>>> a('cat', 'dog')
nice cat dog

>>> b = partial(aa, 'nice', 'banana')
>>> b('apple')
nice banana apple
```

```
def partial(fn, *pargs, **pkwargs):
    print(f'--- wrap fn with {pargs} and {pkwargs}')
    def wrapper(*args, **kwargs):
        print(f'--- appending {args} to {pargs}')
        print(f'--- updating {pkkwargs} with {kwargs}')
        pkwargs.update(kwargs)
        return fn(*(pargs + args), **pkwargs)
    return wrapper

def print3(a, b, c):
    print(a, b, c)
```

```
>>> foo = partial(print3, 'nice')
--- wrap fn with ('nice',) and {}

>>> foo('cat', 'dog')
--- appending ('cat', 'dog') to ('nice',)
--- updating {} with {}
nice cat dog
```

```
def print3(a, b, c):
    print(a, b, c)
```

```
>>> bar = partial(print3, 'nice', 'banana')
--- wrap fn with ('nice', 'banana') and {}

>>> bar('apple')
--- appending ('apple',) to ('nice', 'banana')
--- updating {} with {}
nice banana apple
```

```
>>> baz = partial(print3, c='dog')
--- wrap fn with () and {'c': 'dog'}

>>> baz('cat', b='banana')
--- appending ('cat', 'banana') to ()
--- updating {'c': 'dog'} with {'b': 'banana'}
cat banana dog
```

Nice!

- Simplify function signatures
- Using functions that take >1 arguments with map:
    - map function operand can only accept the current item as argument
- Python actually has such a builtin function

partial

Simplify a function by “freezing” some of its arguments with the supplied values

```
def tomato(taste, color, size):
    print("taste is {}, color is {}, size is {}".format(taste,
                                                        color,
                                                        size))
```

```
>>> from functools import partial

>>> sweet_red_tomato = partial(tomato, color="red", taste="sweet")
>>> sweet_red_tomato(size=7)
taste is sweet, color is red, size is 7
```

printlines

```
>>> printlines = partial(print, sep='\n')

>>> printlines(*['orange', 'grapefruit', 'mango'])
orange
grapefruit
mango
```

map + partial

```
def add(x, y):
    return x + y
```

```
>>> list(map(lambda x: add(x, 7), [1, 2, 3]))
[8, 9, 10]


>>> add7 = partial(add, 7)

>>> list(map(add7, [1, 2, 3]))
>>> [8, 9, 10]
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Decorator functions

- Functions that:
- get a function as an argument
- return a new function wrapping the original one
- add some functionality

```
def counter(fn):
    def wrapper():
        wrapper.count += 1
        res = fn()
        print("+++ fn call counter is {}".format(wrapper.count))
        return res
    wrapper.count = 0
    return wrapper

def cookie():
    print("cookie")

>>> count_cookies = counter(cookie)
>>> count_cookies()
cookie
+++ fn call counter is 1

>>> count_cookies()
cookie
+++ fn call counter is 2

>>> count_cookies()
cookie
+++ fn call counter is 3
```

Logging decorator

```
def logme(fn):
    def internal(*args, **kwargs):
        print('--- calling {}'.format(fn.__name__))
        return fn(*args, **kwargs)
    return internal

def print_3_words(a='first', b='second', c='third'):
    print(a, b, c)

>>> print_3_words()
first second third

>>> print_3_words = logme(print_3_words)
>>> print_3_words()
--- calling print_3_words
first second third
```

Decorator syntax

```
def logme(fn):
    def internal(*args, **kwargs):
        print("--- calling {}".format(fn.__name__))
         return fn(*args, **kwargs)
    return internal

@logme
def print_word(word="word"):
    print(word)

>>> print_word("kitchen")
--- calling print_word
kitchen
```

Loosing function attributes

```
def logme(fn):
    def internal(*args, **kwargs):
        print(f"--- calling {fn.__name__}")
        return fn(*args, **kwargs)
    return internal

def print_word(word="word"):
    """simple function"""
    print(word)

@logme
def print_word_decorated(word="word"):
    """simple function"""
    print(word)

>>> for func in [print_word, print_word_decorated]:
...     print(f"{func.__name__}:\t{func.__doc__}")
print_word: simple function
internal:   None
```

Problem:

A decorated (wrapped) function gets its name and docstring replaced by the wrapper's

Solution:

a doppelgänger decorator,
`@wraps`

Dante Gabriel Rossetti - How They Met Themselves (1864)

```python
from functools import wraps

def logme(fn):
    @wraps(fn)  # duplicate fn's attributes
    def internal(*args, **kwargs):
        print(f"--- calling {fn.__name__}")
        return fn(*args, **kwargs)
    return internal

def print_word(word="word"):
    """simple function"""
    print(word)

@logme
def print_word_decorated(word="word"):
    """simple function"""
    print(word)
```

```
>>> for func in [print_word, print_word_decorated]:
...     print(f"{func.__name__}:\t\t{func.__doc__}")
print_word:             simple function
print_word_decorated: simple function
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

Lab

2c - Closures and decorators

Summary

- Python functions are powerful and flexible

- Python support functional programming via builtin and standard library tools

QUESTION
EVERYTHING

Thanks!

twitter @amitkot
www amitkot.com