Practical Python

Basic Programming in Python

Sebastian Höffner Aline Vilks

Wed, 24 May 2017

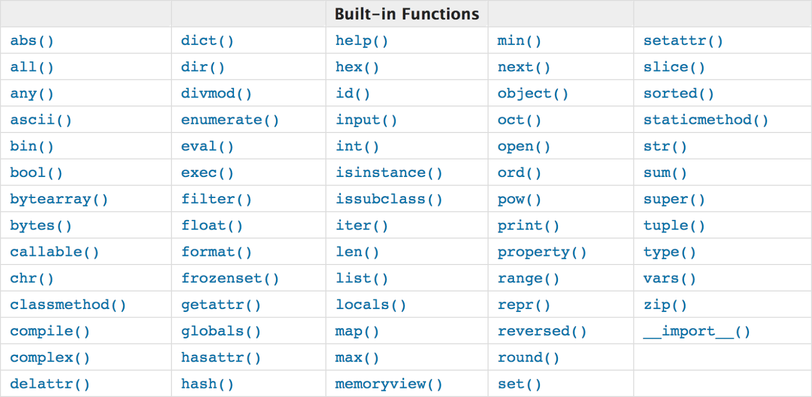
Built-in functions for your everyday tasks

We already discussed some built-in functions[1], for example:

- `open`: Opens a file
- `str`, `float`, `int`: Casts data to the respective types
- `range`: Generates a sequence of numbers
- `enumerate`: Gives us indices and items for iterations
- `set`, `list`, `tuple`, `dict`: Create the corresponding collections

[1] https://docs.python.org/3/library/functions.html

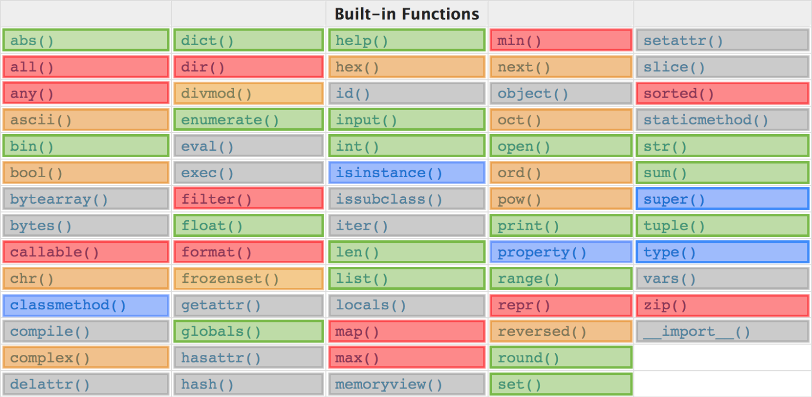
Built-in functions



		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Figure 1: Built-in Functions. (Python Software Foundation 2017)

Built-in functions



		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Figure 2: Green: You know these. Orange: Cover these on your own. Red: Today! Blue: Future sessions. Grey: We won't need these. (Python Software Foundation 2017)

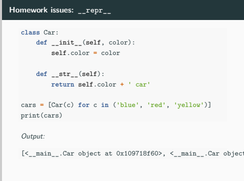
Homework issues: `__repr__`

```python
class Car:
    def __init__(self, color):
        self.color = color

    def __str__(self):
        return self.color + ' car'

cars = [Car(c) for c in ('blue', 'red', 'yellow')]
print(cars)
```

Output:

```
[<__main__.Car object at 0x109718f60>, <__main__.Car object
```

└Homework issues: __repr__



Homework issues: __repr__

```
class Car:
    def __init__(self, color):
        self.color = color

    def __str__(self):
        return self.color + ' car'

cars = [Car(c) for c in ('blue', 'red', 'yellow')]
print(cars)
```

Output:

```
[<__main__.Car object at 0x109718f60>, <__main__.Car object
```

The print functions tries to call `__str__` for all objects you give it. Here, the object is a list! The list's `__str__` function calls its elements' `__repr__` functions.

Homework issues: `__repr__`

`__repr__` should return a string which can be used to create an object which is similar:

```python
class Car:
    def __init__(self, color):
        self.color = color

    def __str__(self):
        return self.color + ' car'

    def __repr__(self):
        return 'Car("' + self.color + '")'

cars = [Car(c) for c in ('blue', 'red', 'yellow')]
print(cars)
```

Output:

```
[Car("blue"), Car("red"), Car("yellow")]
```

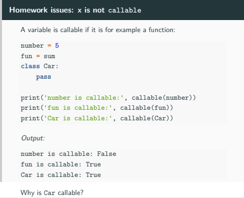
Homework issues: x is not `callable`

A variable is callable if it is for example a function:

```python
number = 5
fun = sum
class Car:
    pass

print('number is callable:', callable(number))
print('fun is callable:', callable(fun))
print('Car is callable:', callable(Car))
```

Output:

```
number is callable: False
fun is callable: True
Car is callable: True
```



Homework issues: x is not callable

A variable is callable if it is for example a function:

```
number = 5
fun = sum
class Car:
    pass

print('number is callable:', callable(number))
print('fun is callable:', callable(fun))
print('Car is callable:', callable(Car))
```

Output:

```
number is callable: False
fun is callable: True
Car is callable: True
```

Why is Car callable?

Car is callable since calling a class (Car()) is creating a new instance.

Homework issues: * (tuple unpacking)

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

print(add(*[1, 2]))
```

Output:

```
3
```

`add(*[1, 2])` is equivalent to `add(1, 2)` – Python “unpacks” the values into each function argument.

General questions: `if __name__ == '__main__':`

- Modules have `__name__`s, the one you run `__main__`, others their file or directory names (without `.py`).
- `import` executes files
- To avoid random prints etc. on import, “secure” your code in `if` block:
  - `if __name__ == '__main__':`
- For extra karma you can put every code in that block into a function (usually `main`):
  - `def main():`
  - Call `main` inside the `if` block
  - This avoids global scope *pollution*

Find the lowest number

```
vacation_offers = [1023.43, 983.4, 985.12, 1014.52]
```

Find the lowest number

```
vacation_offers = [1023.43, 983.4, 985.12, 1014.52]
low = float('inf')
for offer in vacation_offers:
    if offer < low:
        low = offer
print(low)
```

Output:

```
983.4
```

Find the highest number

```
vacation_offers = [1023.43, 983.4, 985.12, 1014.52]
high = -float('inf')
for offer in vacation_offers:
    if offer > high:
        high = offer
print(high)
```

Output:

```
1023.43
```

Python can do it already!

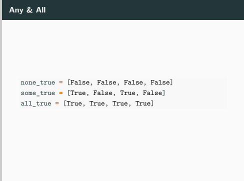
```
vacation_offers = [1023.43, 983.4, 985.12, 1014.52]
print(min(vacation_offers))
print(max(vacation_offers))
```

Output:

```
983.4
1023.43
```

Any & All

```python
none_true = [False, False, False, False]
some_true = [True, False, True, False]
all_true = [True, True, True, True]
```



Any & All

```
none_true = [False, False, False, False]
some_true = [True, False, True, False]
all_true = [True, True, True, True]
```

A very common operation is to check if some values fulfill some condition, all match it, or none.

Later we will see how we can easily create lists of boolean values like the ones above.

Any & All

```python
none_true = [False, False, False, False]
some_true = [True, False, True, False]
all_true = [True, True, True, True]

def any_true(tocheck):
    for elem in tocheck:
        if elem:
            return True
    return False

def all_true(tocheck):
    for elem in tocheck:
        if not elem:
            return False
    return True

print('Any in none?', any_true(none_true))
print('Any in some?', any_true(some_true))
print('All in some?', all_true(some_true))
print('All in all?', all_true(all_true))
```

Any & All

```
none_true = [False, False, False, False]
some_true = [True, False, True, False]
all_true = [True, True, True, True]

print('Any in none?', any(none_true))
print('Any in some?', any(some_true))
print('All in some?', all(some_true))
print('All in all?', all(all_true))
```

Sorting in Python

```python
sorted_list = sorted([9, 2, 5, 3, 1, 8, 19])
print(sorted_list)
sorted_list = sorted([9, 2, 5, 3, 1, 8, 19], reverse=True)
print(sorted_list)
```

Output:

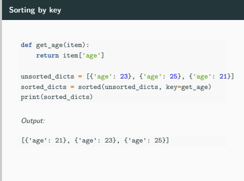
```
[1, 2, 3, 5, 8, 9, 19]
[19, 9, 8, 5, 3, 2, 1]
```

Sorting by key

```python
def get_age(item):
    return item['age']

unsorted_dicts = [{'age': 23}, {'age': 25}, {'age': 21}]
sorted_dicts = sorted(unsorted_dicts, key=get_age)
print(sorted_dicts)
```

Output:

```
[{'age': 21}, {'age': 23}, {'age': 25}]
```



Sorting by key

```python
def get_age(item):
    return item['age']

unsorted_dicts = [{'age': 23}, {'age': 25}, {'age': 21}]
sorted_dicts = sorted(unsorted_dicts, key=get_age)
print(sorted_dicts)
```

Output:

```
[{'age': 21}, {'age': 23}, {'age': 25}]
```

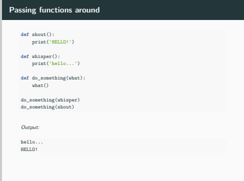
If you attempted the difficult bonus exercise last week, you already saw how to use a key function. Now we will shed some light into it.

Passing functions around

```python
def shout():
    print('HELLO!')

def whisper():
    print('hello...')

def do_something(what):
    what()

do_something(whisper)
do_something(shout)
```

Output:

```
hello...
HELLO!
```



Passing functions around

```
def shout():
    print('HELLO!')

def whisper():
    print('hello...')

def do_something(what):
    what()

do_something(whisper)
do_something(shout)
```

Output:

```
hello...
HELLO!
```

Python always passes by *object reference*. For some objects, those which are mutable, this means that we get references to those objects which we can use and modify. For others, like integers and strings (which are immutable) they get copied themselves.

Mutable objects

```python
def mutate(some_list):
    some_list.append(1)

my_list = []
mutate(my_list)
mutate(my_list)
print(my_list)
```

Output:

```
[1, 1]
```

No reassignment possible

```python
def cantreassign(some_list):
    some_list = [1, 2, 3]

my_list = []
cantreassign(my_list)
print(my_list)
```

Output:

```
[]
```

Using function objects: `map` and `filter`

Python has two interesting functions: `map` and `filter`

Both take two arguments: A function, and an iterable (e.g. a list, a string, . . .)

map

map calls the passed function on each element and stores the results into a map object. This can be transformed into a list:

```
def square(x):
    return x * x

in_list = [1, 2, 3, 4, 5]
out_list = list(map(square, in_list))
print(out_list)
```

Output:

```
[1, 4, 9, 16, 25]
```

filter

filter calls the passed function on each element and stores those elements, for which the result is not False, into a filter object. This can be transformed into a list.

```python
def is_even(x):
    return not x & 1


in_list = [1, 2, 3, 4, 5]
out_list = list(filter(is_even, in_list))
print(out_list)
```

Output:

```
[2, 4]
```

map and filter

Chaining is possible (even without explicit list conversions in between):

```python
def is_even(x):
    return not x & 1

def square(x):
    return x * x

in_list = [1, 2, 3, 4, 5]
out_list = list(map(square, filter(is_even, in_list)))
print(out_list)
```

Output:

```
[4, 16]
```

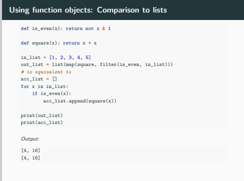
Using function objects: Comparison to lists

```python
def is_even(x): return not x & 1

def square(x): return x * x

in_list = [1, 2, 3, 4, 5]
out_list = list(map(square, filter(is_even, in_list)))
# is equivalent to
acc_list = []
for x in in_list:
    if is_even(x):
        acc_list.append(square(x))

print(out_list)
print(acc_list)
```

Output:

```
[4, 16]
[4, 16]
```

└Using function objects: Comparison to lists



Using function objects: Comparison to lists

```python
def is_even(x): return not x & 1

def square(x): return x * x

in_list = [1, 2, 3, 4, 5]
out_list = list(map(square, filter(is_even, in_list)))
# is equivalent to
acc_list = []
for x in in_list:
    if is_even(x):
        acc_list.append(square(x))

print(out_list)
print(acc_list)
```

Output:

```
[4, 16]
[4, 16]
```

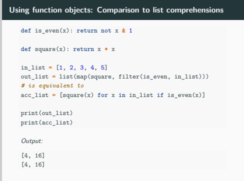
Don’t write functions like this, I just save some space.

Using function objects: Comparison to list comprehensions

```python
def is_even(x): return not x & 1

def square(x): return x * x

in_list = [1, 2, 3, 4, 5]
out_list = list(map(square, filter(is_even, in_list)))
# is equivalent to
acc_list = [square(x) for x in in_list if is_even(x)]

print(out_list)
print(acc_list)
```

Output:

```
[4, 16]
[4, 16]
```



Using function objects: Comparison to list comprehensions

```
def is_even(x): return not x & 1

def square(x): return x * x

in_list = [1, 2, 3, 4, 5]
out_list = list(map(square, filter(is_even, in_list)))
# is equivalent to
acc_list = [square(x) for x in in_list if is_even(x)]

print(out_list)
print(acc_list)
```

Output:

```
[4, 16]
[4, 16]
```

You can read up a little bit more about how to unroll list comprehensions here: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

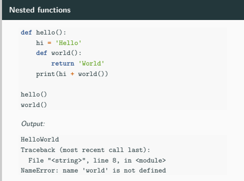
Take a look at the for loop inside the for loop for a hint for the homework ;-)

Nested functions

```python
def hello():
    hi = 'Hello'
    def world():
        return 'World'
    print(hi + world())

hello()
world()
```

Output:

```
HelloWorld
Traceback (most recent call last):
  File "<string>", line 8, in <module>
NameError: name 'world' is not defined
```



Nested functions

```
def hello():
    hi = 'Hello'
    def world():
        return 'World'
    print(hi + world())

hello()
world()
```

Output:

```
HelloWorld
Traceback (most recent call last):
  File "<string>", line 8, in <module>
NameError: name 'world' is not defined
```

Functions are just normal variables, so it's even possible to nest them, i.e. having function declarations inside of function declarations.

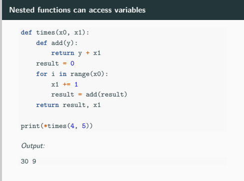
They are only available inside the scope they were declared (except for when you return them and use them somewhere else).

Nested functions can access variables

```python
def times(x0, x1):
    def add(y):
        return y + x1
    result = 0
    for i in range(x0):
        x1 += 1
        result = add(result)
    return result, x1

print(*times(4, 5))
```

Output:

```
30 9
```



Nested functions can access variables

```python
def times(x0, x1):
    def add(y):
        return y + x1
    result = 0
    for i in range(x0):
        x1 += 1
        result = add(result)
    return result, x1

print(*times(4, 5))
```

Output:

```
30 9
```

They can access variables inside the scope they were declared.

In the example, the result is 30 and 9 because:

- `range(4)` has 4 values
- `x1` is incremented in each of the four iterations *before* doing the addition
- `x1` thus takes the values: 6, 7, 8, 9.
- $6 + 7 + 8 + 9 = 30$.

You can return nested functions

```python
def create_adder():
    def adder(x, y):
        return x + y
    return adder

my_add = create_adder()
print(my_add(5, 7))
```

Output:

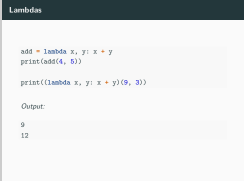
```
12
```

Lambdas

```python
add = lambda x, y: x + y
print(add(4, 5))

print((lambda x, y: x + y)(9, 3))
```

Output:

```
9
12
```



Lambdas

```python
add = lambda x, y: x + y
print(add(4, 5))

print((lambda x, y: x + y)(9, 3))
```

Output:

```
9
12
```

You have seen that it's possible to pass functions around.

This is cool, but sometimes you don't want them to have names and clutter your scope or you feel like this is not a function worth reusing much.

This is where lambdas come into play: small anonymous functions.

They work like normal functions but are slightly limited:

- They don't have a name
- They can only have one statement (which is automatically the return statement)

Why nested functions and lambdas?

- Nested functions and lambdas are used as simple functions for e.g. the `sorted`'s `key` argument.
- They are often used to be passed around.
- They allow *inline* specification of functions you don't really feel worth to be proper functions, e.g. adding two values or combining them into tuples.

`zip`

One powerful functions is `zip`.

Often you will that you have some data which looks like this:

`[(x0, y0), (x1, y1), (x2, y2)]` or `[(x0, y0, z0), (x1, y1, z1), (x2, y2, z3)]`

Or sometimes it will be separate lists:

`[x0, x1, x2]`, `[y0, y1, y2]`, and `[z0, z1, z2]`.

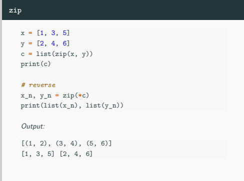
And of course, your favorite plotting library always takes it the other way.

zip

```
x = [1, 3, 5]
y = [2, 4, 6]
c = list(zip(x, y))
print(c)

# reverse
x_n, y_n = zip(*c)
print(list(x_n), list(y_n))
```

Output:

```
[(1, 2), (3, 4), (5, 6)]
[1, 3, 5] [2, 4, 6]
```



zip

```
x = [1, 3, 5]
y = [2, 4, 6]
c = list(zip(x, y))
print(c)

# reverse
x_n, y_n = zip(*c)
print(list(x_n), list(y_n))
```

Output:

```
[(1, 2), (3, 4), (5, 6)]
[1, 3, 5] [2, 4, 6]
```

`zip` works like a zipper. If you have to sides of a zipper `[1, 3, 5]` and `[2, 4, 6]` it will create pairs of those *tooth* which belong together: `list(zip([1, 3, 5], [2, 4, 6]))` results in `[(1, 2), (3, 4), (5, 6)]`.

It is generalized to higher dimensions: If you have *n* lists with *m* elements, you will get one list with *m* tuples containing *n* elements – always the matching ones. That means the `i`-th element of all *n* lists will be inside the `i`-th tuple.

Using tuple unpacking (twice, once to pass the arguments and once implicitly using the return values) you can reverse the process.

zip in higher dimensions

```
x = [1, 4, 7]
y = [2, 5, 8]
z = [3, 6, 9]
c = list(zip(x, y, z))
print(c)

# reverse
x_n, y_n, z_n = zip(*c)
print(list(x_n), list(y_n), list(z_n))
```

Output:

```
[(1, 2, 3), (4, 5, 6), (7, 8, 9)]
[1, 4, 7] [2, 5, 8] [3, 6, 9]
```

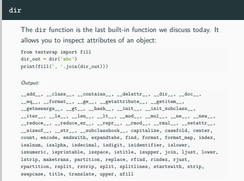
dir

The dir function is the last built-in function we discuss today. It allows you to inspect attributes of an object:

```python
from textwrap import fill
dir_out = dir('abc')
print(fill(', '.join(dir_out)))
```

Output:

```
__add__, __class__, __contains__, __delattr__, __dir__, __doc__,
__eq__, __format__, __ge__, __getattribute__, __getitem__,
__getnewargs__, __gt__, __hash__, __init__, __init_subclass__,
__iter__, __le__, __len__, __lt__, __mod__, __mul__, __ne__, __new__,
__reduce__, __reduce_ex__, __repr__, __rmod__, __rmul__, __setattr__,
__sizeof__, __str__, __subclasshook__, capitalize, casefold, center,
count, encode, endswith, expandtabs, find, format, format_map, index,
isalnum, isalpha, isdecimal, isdigit, isidentifier, islower,
isnumeric, isprintable, isspace, istitle, isupper, join, ljust, lower,
lstrip, maketrans, partition, replace, rfind, rindex, rjust,
rpartition, rsplit, rstrip, split, splitlines, startswith, strip,
swapcase, title, translate, upper, zfill
```



dir

The dir function is the last built-in function we discuss today. It allows you to inspect attributes of an object:

```
from textwrap import fill
dir_out = dir('abc')
print(fill(', '.join(dir_out)))
```

Output:

```
__add__, __class__, __contains__, __delattr__, __dir__, __doc__,
__eq__, __format__, __ge__, __getattribute__, __getitem__,
__getnewargs__, __gt__, __hash__, __init__, __init_subclass__,
__iter__, __le__, __len__, __lt__, __mod__, __mul__, __ne__, __new__,
__reduce__, __reduce_ex__, __repr__, __rmod__, __rmul__, __setattr__,
__sizeof__, __str__, __subclasshook__, capitalize, casefold, center,
count, encode, endswith, expandtabs, find, format, format_map, index,
isalnum, isalpha, isdecimal, isdigit, isidentifier, islower,
isnumeric, isprintable, isspace, istitle, isupper, join, ljust, lower,
lstrip, maketrans, partition, replace, rfind, rindex, rjust,
rpartition, rsplit, rstrip, split, splitlines, startswith, strip,
swapcase, title, translate, upper, zfill
```

While this is not really something you use in practice, it allows you to debug some of your programs or to get some ideas of what might be available for your objects.

In the example you can see many functions and attributes str objects have.

Your eighth homework

Today we discussed the differences between

- `map`, `filter`, `lambda` (and other functions)
- lists with accumulators
- list comprehensions
- Implement some simple lists using all of the above methods to get an idea of how to transform between them and which are more appropriate in which situation.
- Use a custom function to sort cars by their comfort.

References

Python Software Foundation. 2017. *Python 3.6.0 Documentation*. 3.6.0 ed. Beaverton, Oregon, USA: Python Software Foundation.