First Class Functions:

Functions in python is what we call a first-class object. A first-class object id a program entity that can be:

* Created at runtime
* Assigned to a variable or element in a data structure
* Passed as an argument to a function
* Returned as the result of a function

Integers, strings, and dictionaries are other examples of first-class objects in Python— nothing fancy here. The term “first-class functions” is widely used as shorthand for “functions as first-class objects.” It’s not perfect because it seems to imply an “elite” among functions. In Python, all function are first-class.

Treating a Function Like an Object:

The console session in Example 5-1 shows that Python functions are objects. Here we create a function, call it, read its \_\_doc\_\_ attribute, and check that the function object itself is an instance of the function class.

Create and test a function, then read its \_\_doc\_\_ and check its type

>>> def factorial(n):

... '''returns n!'''

... return 1 if n < 2 else n \* factorial(n-1)

...

>>> factorial(42)

1405006117752879898543142606244511569936384000000000

>>> factorial.\_\_doc\_\_

'returns n!'

>>> type(factorial)

<class 'function'>

>>>

Here we are using a console, this means that this is happening at runtime, meaning, that we can then define a function as a first-class object. The \_\_doc\_\_ attribute is used to generate the help text of an object. In the Python interactive console, the command help(factorial).

Higher-Order Functions:

A function that takes a function as argument or returns a function as the result is a *higher-order function*. One example is map. Another is the built-in function sorted: an optional key argument lets you provide a function to be applied to each item for sorting:

Sorting a list of words by length:

>>>

>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']

>>> sorted(fruits, key=len)

['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']

>>>

In the functional programming paradigm, some of the best known higher-order functions are map, filter, reduce, and apply. The apply function was deprecated in Python 2.3 and removed in Python 3 because it’s no longer necessary. If you need to call a function with a dynamic set of arguments, you can just write fn(\*args, \*\*key words) instead of apply(fn, args, kwargs). The map, filter, and reduce higher-order functions are still around, but better alter‐ natives are available for most of their use cases.

Modern Replacements for map, filter, and reduce:

Functional languages commonly offer the map, filter, and reduce higher-order functions (sometimes with different names). The map and filter functions are still builtins in Python 3, but since the introduction of list comprehensions and generator expressions, they are not as important. A listcomp or a genexp does the job of map and filter combined but is more readable.

Lists of factorials produced with map and filter compared to alternatives coded as list comprehensions:

>>> list(map(fact, range(6)))

[1, 1, 2, 6, 24, 120]

>>> [fact(n) for n in range(6)]

[1, 1, 2, 6, 24, 120]

>>> list(map(factorial, filter(lambda n: n % 2, range(6))))

[1, 6, 120]

>>> [factorial(n) for n in range(6) if n % 2]

[1, 6, 120]

>>>

In Python 3, map and filter return generators—a form of iterator—so their direct substitute is now a generator expression (in Python 2, these functions returned lists, therefore their closest alternative is a listcomp). The reduce function was demoted from a built-in in Python 2 to the functools module in Python 3. Its most common use case, summation, is better served by the sum built-in available since Python 2.3 was released in 2003. This is a big win in terms of readability and performance

Sum of integers up to 99 performed with reduce and sum:

>>> from functools import reduce

>>> from operator import add

>>> reduce(add, range(100)) [1, 6, 120]

4950

>>> sum(range(100))

4950

>>>

The common idea of sum and reduce is to apply some operation to successive items in a sequence, accumulating previous results, thus reducing a sequence of values to a single value.

Anonymous Functions:

The lambda keyword creates an anonymous function within a Python expression. However, the simple syntax of Python limits the body of lambda functions to be pure expressions. In other words, the body of a lambda cannot make assignments or use any other Python statement such as while, try, etc. The best use of anonymous functions is in the context of an argument list.

Sorting a list of words by their reversed spelling using lambda:

>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']

>>> sorted(fruits, key=lambda word: word[::-1])

['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']

>>>

Outside the limited context of arguments to higher-order functions, anonymous func‐ tions are rarely useful in Python. The syntactic restrictions tend to make nontrivial lambdas either unreadable or unworkable.

Lundh’s lambda Refactoring Recipe

If you find a piece of code hard to understand because of a lambda, Fredrik Lundh suggests this refactoring procedure:

1. Write a comment explaining what the heck that lambda does.

2. Study the comment for a while, and think of a name that captures the essence of

the comment.

3. Convert the lambda to a def statement, using that name.

4. Remove the comment.

The lambda syntax is just syntactic sugar: a lambda expression creates a function object just like the def statement. That is just one of several kinds of callable objects in Python. The following section reviews all of them.

The Seven Flavors of Callable Objects:

The call operator (i.e., ()) may be applied to other objects beyond user-defined functions. To determine whether an object is callable, use the callable() built-in function. The Python Data Model documentation lists seven callable types:

User-defined functions

Created with def statements or lambda expressions.

Built-in functions

A function implemented in C (for CPython), like len or time.strftime.

Built-in methods

Methods implemented in C, like dict.get.

Methods

Functions defined in the body of a class.

Classes

When invoked, a class runs its \_\_new\_\_ method to create an instance, then \_\_init\_\_ to initialize it, and finally the instance is returned to the caller. Because there is no new operator in Python, calling a class is like calling a function. (Usually calling a class creates an instance of the same class, but other behaviors are possible by overriding \_\_new\_\_.

Class instances

If a class defines a \_\_call\_\_ method, then its instances may be invoked as functions.

Generator functions

Functions or methods that use the yield keyword. When called, generator functions return a generator object.

User-Defined Callable Types:

Not only are Python functions real objects, but arbitrary Python objects may also be made to behave like functions. Implementing a \_\_call\_\_ instance method is all it takes. The following example implements a BingoCage class. An instance is built from any iterable, and stores an internal list of items, in random order. Calling the instance pops an item.

import random

class BingoCage:

    def \_\_init\_\_(self, items):

        self.\_items = list(items)

        random.shuffle(self.\_items)

    def pick(self):

        try:

            return self.\_items.pop()

        except IndexError:

            raise LookupError('pick from empty BingoCage')

    def \_\_call\_\_(self):

        return self.pick()

Here is a demo of the code written before:

>>> bingo = BingoCage(range(3))

>>> bingo.pick()

1

>>> bingo()

0

>>> callable(bingo)

True

>>>

A class implementing \_\_call\_\_ is an easy way to create function-like objects that have some internal state that must be kept across invocations, like the remaining items in the BingoCage. Also, we can see that the \_\_call\_\_ function we have implemented is actually a higher order function, because it’s actually returning another function. An example is a decorator. Decorators must be functions, but it is sometimes convenient to be able to “remember” something between calls of the decorator.

Function Introspection:

Function objects have many attributes beyond \_\_doc\_\_. See what the dir() function reveals about our factorial:

>>>

>>> dir(factorial)

['\_\_annotations\_\_', '\_\_call\_\_', '\_\_class\_\_', '\_\_closure\_\_', '\_\_code\_\_',

'\_\_defaults\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_','\_\_format\_\_', '\_\_ge\_\_', '\_\_get\_\_', '\_\_getattribute\_\_', '\_\_globals\_\_',

'\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_kwdefaults\_\_', '\_\_le\_\_', '\_\_lt\_\_',

'\_\_module\_\_', '\_\_name\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_qualname\_\_', '\_\_reduce\_\_',

'\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_',

'\_\_subclasshook\_\_']

>>>

Most of these attributes are common to Python objects in general. In this section, we cover those that are especially relevant to treating functions as objects, starting with \_\_dict\_\_.

Listing attributes of functions that don’t exist in plain instances:

>>> class C: pass #

>>> obj = C() #

>>> def func(): pass #

>>> sorted(set(dir(func)) - set(dir(obj))) #

['\_\_annotations\_\_', '\_\_call\_\_', '\_\_closure\_\_', '\_\_code\_\_', '\_\_defaults\_\_',

'\_\_get\_\_', '\_\_globals\_\_', '\_\_kwdefaults\_\_', '\_\_name\_\_', '\_\_qualname\_\_']

>>>

Here we create an object, and a function, and then we see the output of the difference of the set of the elements contained in a primary function, and in a primary object from a random class.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| \_\_annotations\_\_ | dict | Parameter and return annotations |
| \_\_call\_\_ | method-wrapper | Implementation of the () operator; a.k.a. the callable object protocol |
| \_\_closure\_\_ | tuple | The function closure, i.e., bindings for free variables (often is None) |
| \_\_code\_\_ | code | Function metadata and function body compiled into bytecode |
| \_\_defaults\_\_ | tuple | Default values for the formal parameters |
| \_\_get\_\_ | method-wrapper | Implementation of the read-only descriptor protocol |
| \_\_globals\_\_ | dict | Global variables of the module where the function is defined |
| \_\_kwdefaults\_\_ | dict | Default values for the keyword-only formal parameters |
| \_\_name\_\_ | str | The function name |
| \_\_qualname\_\_ | str | The qualified function name, e.g., Random.choice |

We will discuss the \_\_defaults\_\_, \_\_code\_\_, and \_\_annotations\_\_ functions, used by IDEs and frameworks to extract information about function signatures, in later sections. But to fully appreciate these attributes, we will make a detour to explore the powerful syntax Python offers to declare function parameters and to pass arguments into them.

From Positional to Keyword-Only Parameters:

One of the best features of Python functions is the extremely flexible parameter handling mechanism, enhanced with keyword-only arguments in Python 3. Closely related are the use of \* and \*\* to “explode” iterables and mappings into separate arguments when we call a function.

The following example is a super helpful one because it can actually take a function in python and translate it to html by using the elements used to describe the code in the HTML file by passing different arguments. Note that a keyword-only argument cls is used to pass “class” attributes as a workaround because class is a keyword in Python

Keyword-only arguments are a new feature in Python 3. In Example 5-10, the cls parameter can only be given as a keyword argument—it will never capture unnamed positional arguments. To specify keyword-only arguments when defining a function, name them after the argument prefixed with \*. If you don’t want to support variable positional arguments but still want keyword-only arguments, put a \* by itself in the signature, like this:

>>> def f(a, \*, b):

... return a, b

>>> f(1, b=2)

(1, 2)

The tag function generates HTML:

def tag(name, \*content, cls=None, \*\*attrs):

    """Generate one or more HTML tags"""

    if cls is not None:

        attrs['class'] = cls

    if attrs:

        attr\_str = ''.join(' %s="%s"' % (attr, value) for attr, value in sorted(attrs.items()))

    else:

        attr\_str = ''

    if content:

        return '\n'.join('<%s%s>%s</%s>' % (name, attr\_str, c, name) for c in content)

    else:

        return '<%s%s />' % (name, attr\_str)

>>> tag('br')

'<br />'

>>> tag('p', 'hello')

'<p>hello</p>'

>>> print(tag('p', 'hello', 'world'))

<p>hello</p>

<p>world</p>

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

>>> tag('p', 'hello', id=33)

'<p id="33">hello</p>'

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

>>> print(tag('p', 'hello', 'world', cls='sidebar'))

<p class="sidebar">hello</p>

<p class="sidebar">world</p>

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

>>> tag(content='testing', name="img")

'<img content="testing" />'

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

>>> my\_tag = {'name': 'img', 'title': 'Sunset Boulevard',

... 'src': 'sunset.jpg', 'cls': 'framed'}

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

>>> tag(\*\*my\_tag)

'<img class="framed" src="sunset.jpg" title="Sunset Boulevard" />'

Retrieving Information About Parameters:

There is a framework called bobo, this one can be passed as a decorator to a function in order to generate a http response with an html text. If your function takes a variable to return the string, let’s say a name, to return “Hello” and after the name of the person then you will get a text on the shell letting you know which variable is the one that is missing, and also you’ll get a 403-error code. The if you pass the variable name it will work. The deal here is:

How did bobo know which variable was missing?

Within a function object, the \_\_defaults\_\_ attribute holds a tuple with the default values of positional and keyword arguments. The defaults for keyword-only arguments appear in \_\_kwdefaults\_\_. The names of the arguments, however, are found within the \_\_code\_\_ attribute, which is a reference to a code object with many attributes of its own.

Let’s create a new function and then use the methods:

def clip(text, max\_len=80):

    """Return text clipped at the last space before or after max\_len"""

    end = None

    if len(text) > max\_len:

        space\_before = text.rfind(' ', 0, max\_len)

        if space\_before >= 0:

            end = space\_before

        else:

            space\_after = text.rfind(' ', max\_len)

            if space\_after >= 0:

                end = space\_after

    if end is None: # no spaces were found

        end = len(text)

    return text[:end].rstrip()

Here’s how bobo knows:

>>> from clip import clip

>>> clip.\_\_defaults\_\_

>>> (80,)

>>> clip.\_\_code\_\_ # doctest: +ELLIPSIS

<code object clip at 0x...>

>>> clip.\_\_code\_\_.co\_varnames

('text', 'max\_len', 'end', 'space\_before', 'space\_after')

>>> clip.\_\_code\_\_.co\_argcount

2>>>

This way os saving the information about the variables contained inside of the function is not the best, this is because we can get with func.\_\_code\_\_.co\_argcount the number of arguments, then inside the tuple \_\_code\_\_.co\_varnames we have everything mixed, the argument names, and the variable’s names. We can say which ones are arguments and which ones are variables by counting how many arguments the \_\_code\_\_.co\_argcount returns, and then those first names, will be for the arguments, and the rest for actual variables inside the code structure of the function. Also, this does not include the arguments by position prefixed by \* or \*\*. The default values are identified only by their position in the \_\_defaults\_\_ tuple, so to link each with the respective argument, you have to scan from last to first.

This seems like a bit of a problem but luckily we have the inspect module to deal with this:

Extracting the function signature:

>>> from clip import clip

>>> from inspect import signature

>>> sig = signature(clip)

>>> sig # doctest: +ELLIPSIS

<inspect.Signature object at 0x...>

>>> str(sig)

'(text, max\_len=80)'

>>> for name, param in sig.parameters.items():

... print(param.kind, ':', name, '=', param.default)

...

POSITIONAL\_OR\_KEYWORD : text = <class 'inspect.\_empty'>

POSITIONAL\_OR\_KEYWORD : max\_len = 80>>>

This is much better. inspect.signature returns an inspect.Signature object, which has a parameters attribute that lets you read an ordered mapping of names to in spect.Parameter objects. Each Parameter instance has attributes such as name, de fault, and kind. The special value inspect.\_empty denotes parameters with no default, which makes sense considering that None is a valid—and popular—default value. The kind attribute holds one of five possible values from the \_ParameterKind class:

POSITIONAL\_OR\_KEYWORD

A parameter that may be passed as a positional or as a keyword argument (most Python function parameters are of this kind).

VAR\_POSITIONAL

A tuple of positional parameters.

VAR\_KEYWORD

A dict of keyword parameters.

KEYWORD\_ONLY

A keyword-only parameter (new in Python 3).

POSITIONAL\_ONLY

A positional-only parameter; currently unsupported by Python function declaration syntax, but exemplified by existing functions implemented in C—like divmod —that do not accept parameters passed by keyword.

An inspect.Signature object has a bind method that takes any number of arguments and binds them to the parameters in the signature, applying the usual rules for matching actual arguments to formal parameters. This can be used by a framework to validate arguments prior to the actual function invocation.

We’ll explain that in the following example:

Binding the function signature from the tag function to a dict of arguments:

>>>import inspect

>>> sig = inspect.signature(tag)

>>> my\_tag = {'name': 'img', 'title': 'Sunset Boulevard',

... 'src': 'sunset.jpg', 'cls': 'framed'}

>>> bound\_args = sig.bind(\*\*my\_tag)

>>> bound\_args

<inspect.BoundArguments object at 0x...>

>>> for name, value in bound\_args.arguments.items():

... print(name, '=', value)

...

name = img

cls = framed

attrs = {'title': 'Sunset Boulevard', 'src': 'sunset.jpg'}

>>> del my\_tag['name']

>>> bound\_args = sig.bind(\*\*my\_tag)

Traceback (most recent call last):

...

TypeError: 'name' parameter lacking default value

>>>

This example shows how the Python data model, with the help of inspect, exposes the same machinery the interpreter uses to bind arguments to formal parameters in function calls. Frameworks and tools like IDEs can use this information to validate code. Another feature of Python 3, function annotations, enhances the possible uses of this, as we will see next.

Function Annotations:

Python 3 provides syntax to attach metadata to the parameters of a function declaration and its return value. Function annotations only occur in the first line of code, when we are passing the arguments and the parameters. Now let’s see the same function as before, but now with annotations to see how it looks:

def clip(text:str, max\_len:'int > 0'=80) -> str:

    """Return text clipped at the last space before or after max\_len"""

    end = None

    if len(text) > max\_len:

        space\_before = text.rfind(' ', 0, max\_len)

        if space\_before >= 0:

            end = space\_before

        else:

            space\_after = text.rfind(' ', max\_len)

            if space\_after >= 0:

                end = space\_after

    if end is None: # no spaces were found

        end = len(text)

    return text[:end].rstrip()

Each argument in the function declaration may have an annotation expression preceded by :. If there is a default value, the annotation goes between the argument name and the = sign. To annotate the return value, add -> and another expression between the ) and the : at the tail of the function declaration. The expressions may be of any type. The most common types used in annotations are classes, like str or int, or strings, like 'int > 0', as seen in the annotation for max\_len

No processing is done with the annotations. They are merely stored in the \_\_annotations\_\_ attribute of the function, a dict:

>>> from clip\_annot import clip

>>> clip.\_\_annotations\_\_

{'text': <class 'str'>, 'max\_len': 'int > 0', 'return': <class 'str'>}>>>

The item with key 'return' holds the return value annotation marked with -> in the function declaration. The only thing Python does with annotations is to store them in the \_\_annotations\_\_ attribute of the function. Nothing else: no checks, enforcement, validation, or any other action is performed. In other words, annotations have no meaning to the Python interpreter. They are just metadata that may be used by tools, such as IDEs, frameworks, and decorators. At this writing no tools that use this metadata exist in the standard library, except that inspect.signature() knows how to extract the annotations. Of course, we can also extract the annotations with the signature function:

Extracting annotations from the function signature:

>>>from clip\_annot import clip

>>> from inspect import signature

>>> sig = signature(clip)

>>> sig.return\_annotation

<class 'str'>

>>> for param in sig.parameters.values():

... note = repr(param.annotation).ljust(13)

... print(note, ':', param.name, '=', param.default)

<class 'str'> : text = <class 'inspect.\_empty'>

'int > 0' : max\_len = 80>>>

The signature function returns a Signature object, which has a return\_annotation attribute and a parameters dictionary mapping parameter names to Parameter objects. Each Parameter object has its own annotation attribute.

In the future, frameworks such as Bobo could support annotations to further automate request processing. For example, an argument annotated as price:float may be automatically converted from a query string to the float expected by the function; a string annotation like quantity:'int > 0' might be parsed to perform conversion and validation of a parameter.

Packages for Functional Programming:

Although Guido makes it clear that Python does not aim to be a functional programming language, a functional coding style can be used to good extent, thanks to the support of packages like operator and functools.

The operator module:

Often in functional programming it is convenient to use an arithmetic operator as a function. For example, suppose you want to multiply a sequence of numbers to calculate factorials without using recursion. To perform summation, you can use sum, but there is no equivalent function for multiplication. You could use reduce but this requires a function to multiply two items of the sequence, and we don’t have one, so we have to import it from the operator module.

Factorial implemented with reduce and operator.mul:

>>> from functools import reduce

from operator import mul

def fact(n):

return reduce(mul, range(1, n+1))>>>

This could’ve been done with a lambda function, but reading would be a little bit complicated. This then shows how we can sometimes substitute the lambdas for this kind of functions. Another group of one-trick lambdas that operator replaces are functions to pick items from sequences or read attributes from objects: itemgetter and attrgetter actually build custom functions to do that.

The following example shows a common use of itemgetter: sorting a list of tuples by the value of one field. In the example, the cities are printed sorted by country code (field 1). Essentially, itemgetter(1) does the same as lambda fields: fields[1]: create a function that, given a collection, returns the item at index 1.

Demo of itemgetter to sort a list of tuples:

>>> metro\_data = [

... ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),

... ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),

... ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),

... ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),

... ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),

... ]

>>>

>>> from operator import itemgetter

>>> for city in sorted(metro\_data, key=itemgetter(1)):

... print(city)

...

('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833))

('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))

('Tokyo', 'JP', 36.933, (35.689722, 139.691667))

('Mexico City', 'MX', 20.142, (19.433333, -99.133333))

('New York-Newark', 'US', 20.104, (40.808611, -74.020386))>>>

If you pass multiple index arguments to itemgetter, the function it builds will return tuples with the extracted values:

>>> cc\_name = itemgetter(1, 0)

>>> for city in metro\_data:

... print(cc\_name(city))

...

('JP', 'Tokyo')

('IN', 'Delhi NCR')

('MX', 'Mexico City')

('US', 'New York-Newark')

('BR', 'Sao Paulo')

>>>

Because itemgetter uses the [] operator, it supports not only sequences but also mappings and any class that implements \_\_getitem\_\_.

A sibling of itemgetter is attrgetter, which creates functions to extract object at‐ tributes by name. If you pass attrgetter several attribute names as arguments, it also returns a tuple of values. In addition, if any argument name contains a . (dot), attrgetter navigates through nested objects to retrieve the attribute.

Demo of attrgetter to process a previously defined list of namedtuple called metro\_data:

>>> from collections import namedtuple

>>> LatLong = namedtuple('LatLong', 'lat long')

>>> Metropolis = namedtuple('Metropolis', 'name cc pop coord')

>>> metro\_areas = [Metropolis(name, cc, pop, LatLong(lat, long))

... for name, cc, pop, (lat, long) in metro\_data]

>>> metro\_areas[0]

Metropolis(name='Tokyo', cc='JP', pop=36.933, coord=LatLong(lat=35.689722,

long=139.691667))

>>> metro\_areas[0].coord.lat

35.689722

>>> from operator import attrgetter

>>> name\_lat = attrgetter('name', 'coord.lat')

>>>

>>> for city in sorted(metro\_areas, key=attrgetter('coord.lat')):

... print(name\_lat(city))

('Sao Paulo', -23.547778)

('Mexico City', 19.433333)

('Delhi NCR', 28.613889)

('Tokyo', 35.689722)

('New York-Newark', 40.808611)

Here we have created two namedtuple objects, one named Metropolis, containing the other named Latlong. We built the metro\_areas list with instances of the Metropolis namedtuple based on the metro\_data we had before.

Here is a partial list of functions defined in operator (names starting with \_ are omitted, because they are mostly implementation details):

>>>[name for name in dir(operator) if not name.startswith('\_')]

['abs', 'add', 'and\_', 'attrgetter', 'concat', 'contains',

'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',

'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imod', 'imul',

'index', 'indexOf', 'inv', 'invert', 'ior', 'ipow', 'irshift',

'is\_', 'is\_not', 'isub', 'itemgetter', 'itruediv', 'ixor', 'le',

'length\_hint', 'lshift', 'lt', 'methodcaller', 'mod', 'mul', 'ne',

'neg', 'not\_', 'or\_', 'pos', 'pow', 'rshift', 'setitem', 'sub',

'truediv', 'truth', 'xor']>>>

Most of the 52 names listed are self-evident. The group of names prefixed with i and the name of another operator—e.g., iadd, iand, etc.—correspond to the augmented assignment operators—e.g., +=, &=, etc. These change their first argument in place, if it is mutable; if not, the function works like the one without the i prefix: it simply returns the result of the operation.

Of the remaining operator functions, methodcaller is the last we will cover. It is somewhat similar to attrgetter and itemgetter in that it creates a function on the fly. The function it creates calls a method by name on the object given as argument,

Demo of methodcaller: second test shows the binding of extra arguments:

>>> from operator import methodcaller

>>> s = 'The time has come'

>>> upcase = methodcaller('upper')

>>> upcase(s)

'THE TIME HAS COME'

>>> hiphenate = methodcaller('replace', ' ', '-')

>>> hiphenate(s)

'The-time-has-come'

>>>

Freezing Arguments with functools.partial:

The functools module brings together a handful of higher-order functions. The best known of them is probably reduce. Of the remaining functions in functools, the most useful is partial and its variation, partialmethod.

functools.partial is a higher-order function that allows partial application of a function. Given a function, a partial application produces a new callable with some of the arguments of the original function fixed. This is useful to adapt a function that takes one or more arguments to an API that requires a callback with fewer arguments.

Using partial to use a two-argument function where a one-argument callable is required:

>>> from operator import mul

>>> from functools import partial

>>> triple = partial(mul, 3)

>>> triple(7)

21

>>> list(map(triple, range(1, 10)))

[3, 6, 9, 12, 15, 18, 21, 24, 27]>>>

A more useful example involves the unicode.normalize If you work with text from many languages, you may want to apply unicode.normalize('NFC', s) to any string s before comparing or storing it. If you do that often, it’s handy to have an nfc function to do so:

Building a convenient Unicode normalizing function with partial:

>>>

>>> import unicodedata, functools

>>> nfc = functools.partial(unicodedata.normalize, 'NFC')

>>> s1 = 'café'

>>> s2 = 'cafe\u0301'

>>> s1, s2

('café', 'café')

>>> s1 == s2

False

>>> nfc(s1) == nfc(s2)

True>>>

partial takes a callable as first argument, followed by an arbitrary number of positional and keyword arguments to bind.

The functools.partialmethod function (new in Python 3.4) does the same job as partial but is designed to work with methods. An impressive functools function is lru\_cache, which does memorization—a form of automatic optimization that works by storing the results of function calls to avoid ex‐ pensive recalculations.

Summary:

First class nature of functions and high order functions:

The main ideas are that you can assign functions to variables, pass them to other functions, store them in data structures, and access function attributes, allowing frameworks and tools to act on that information. Higher-order functions, a staple of functional programming, are common in Python—even if the use of map, filter, and reduce is not as frequent as it was—thanks to list comprehensions (and similar constructs like generator expressions) and the appearance of reducing built-ins like sum, all, and any. The sorted, min, max built-ins, and functools.partial are examples of commonly used higher-order functions in the language.

Callables:

Callables come in seven different flavors in Python, from the simple functions created with lambda to instances of classes implementing \_\_call\_\_. They can all be detected by the callable() built-in function. Every callable supports the same rich syntax for declaring formal parameters, including keyword-only parameters and annotations—both new features introduced with Python 3.

Annotations:

Python functions and their annotations have a rich set of attributes that can be read with the help of the inspect module, which includes the Signature.bind method to apply the flexible rules that Python uses to bind actual arguments to declared parameters.