Python – S6

Contents

Anonymous Functions: lambda

Anonymous Functions: lambda

- Besides the def statement, Python also provides an expression form that generates function objects.
- Because of its similarity to a tool in the Lisp language, it's called lambda.
- Like def, this expression creates a function to be called later, but it returns the function instead of assigning it to a name.
- This is why lambdas are sometimes known as anonymous (i.e., unnamed) functions.
- In practice, they are often used as a way to inline a function definition, or to defer execution of a piece of code.
- The lambda's general form is the keyword lambda, followed by one or more arguments (exactly like the arguments list you enclose in parentheses in a def header), followed by an expression after a colon:

lambda *argument1*, *argument2*,... *argumentN* : *expression using arguments*

Anonymous Functions: lambda

- Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs, but there are a few differences that make lambdas useful in specialized roles:

- **lambda** is an expression, not a statement.
- Because of this, a lambda can appear in places a def is not allowed by Python's syntax—inside a list literal or a function call's arguments, for example.
- With def, functions can be referenced by name but must be created elsewhere.
- As an expression, lambda returns a value (a new function) that can optionally be assigned a name.
- In contrast, the def statement always assigns the new function to the name in the header, instead of returning it as a result.

Anonymous Functions: lambda

- **lambda**'s body is a single expression, not a block of statements.
- The lambda's body is similar to what you'd put in a def body's return statement; you simply type the result as a naked expression, instead of explicitly returning it.
- Because it is limited to an expression, a lambda is less general than a def—you can only squeeze so much logic into a lambda body without using statements such as if.
- This is by design, to limit program nesting: lambda is designed for coding simple functions, and def handles larger tasks.

Anonymous Functions: lambda

- Apart from those distinctions, defs and lambdas do the same sort of work.
- For instance, we've seen how to make a function with a def statement:

```
>>> def func(x, y, z):
        return x + y + z
>>> func(2, 3, 4)
9
```

- But you can achieve the same effect with a lambda expression by explicitly assigning its result to a name through which you can later call the function:

```
>>> f = lambda x, y, z:  x + y + z
>>> f(2, 3, 4)
9
```

- Here, f is assigned the function object the lambda expression creates; this is how def works, too, but its assignment is automatic.

Anonymous Functions: lambda

- Defaults work on lambda arguments, just like in a def:

```
>>> x = (lambda a="fee", b="fie", c="foe":  a + b + c)
>>> x("wee")
'weefiefoe'
```

- The code in a lambda body also follows the same scope lookup rules as code inside a def.
- lambda expressions introduce a local scope much like a nested def, which automatically sees names in enclosing functions, the module, and the built-in scope via the LEGB rule.

Anonymous Functions: lambda

Why use lambda ?

- Generally speaking, lambda comes in handy as a sort of function shorthand that allows you to embed a function's definition within the code that uses it.
- They are entirely optional—you can always use def instead, and should if your function requires the power of full statements that the lambda's expression cannot easily provide—but they tend to be simpler coding constructs in scenarios where you just need to embed small bits of executable code inline at the place it is to be used.
- lambda is also commonly used to code jump tables, which are lists or dictionaries of actions to be performed on demand. For example:

```
L = [lambda x: x ** 2,
     lambda x: x ** 3,
     lambda x: x ** 4]
for f in L:
     print(f(2))   # prints 4, 8, 16
print(L[0](3))     # prints 9
```

Anonymous Functions: lambda

- So instead of defining a function, Python allows us to provide a lambda form.
- This is a kind of anonymous, one-use-only function body in places where we only need a very, very simple function.
- A lambda form is like a defined function: it has arguments/parameters and computes a value.
- The body of a lambda, however, can only be a single expression, limiting it to relatively simple operations.
- If it gets complex, you'll have to define a real function.
- Note that a lambda form is not a statement; it's an expression that's used within other expressions.
- The lambda form does not define an object with a long life-time.
- The lambda form object – generally – exists just in the scope of a single statement's execution.

Anonymous Functions: lambda

- Generally, a lambda will look like this.

lambda a: a[0]*2 + a[1]

- This is a lambda which takes a tuple argument value and returns a value based on the first two elements of the tuple.

- Consider another example:

```
>>> from math import pi
>>> (lambda x: pi*x*x)(5)
78.539816339744831
```

- Please note:

1. The '(lambda x: pi*x*x)' is a function with no name; it accepts a single argument, x, and computes 'pi*x*x'.
2. We apply the lambda to an argument value of 5.

map and filter functions

- Two built-in functions in Python are quite useful are : map and filter
- For example :

```
>>> list(map(str.upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
```

- The syntax is : map(function, sequence, [...])
  Create a new list from the results of applying the given function to the items of the the given sequence.

```
>>> list(map( int, [ "10", "12", "14", 3.1415926 ] ))
[ 10, 12, 14, 3 ]
```

- If any sequence is too short, None is used for missing the values. If the function is None, map() will create tuples from corresponding items in each list, much like the zip() function.
- Another simple example:

```
>>> list(map(abs, (-1, 0, 1)))
[ 1, 0, 1]
```

map and filter functions

- The syntax of filter function is : filter(function, sequence)
- Return a list containing those items of sequence for which the given function is True.
- If the function is None, return a list of items that are equivalent to True.
- An example :

```
>>> list(filter(None, ['spam', '', 'ni']))
['spam', 'ni']
```

- Another example :

```
>>> def is_even(x):
...         return (x % 2) == 0
>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]
```

- Please note that the above can be written also using list comprehension as :

```
>>> list( x for x in range(10) if is_even(x) )
[0, 2, 4, 6, 8]
```

Functional Programming Tools

- By most definitions, today's Python blends support for multiple programming paradigms: procedural (with its basic statements), object-oriented (with its classes), and functional.
- For the latter of these, Python includes a set of built-ins used for functional programming—tools that apply functions to sequences and other iterables.

- This set includes tools that call functions on an iterable's items (map); filter out items based on a test function (filter); and apply functions to pairs of items and running results (reduce).

- The map function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results.

Functional Programming Tools

- The map function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results.
- For example:

```
>>>  counters = [ 1, 2, 3, 4 ]
def inc(x): return x + 10  # Function to be run
>>>  list(map(inc, counters))   # Collect results
[11, 12, 13, 14]
```

- map calls inc on each list item and collects all the return values into a new list.

Functional Programming Tools

- Because it also returns an iterable, filter (like range) requires a list call to display all its results
- For example, the following filter call picks out items in a sequence that are greater than zero:

```
>>> list(range(−5, 5))
[−5, −4, −3, −2, −1, 0, 1, 2, 3, 4]
>>> list(filter((lambda x: x > 0), range(−5, 5)))
[1, 2, 3, 4]
```

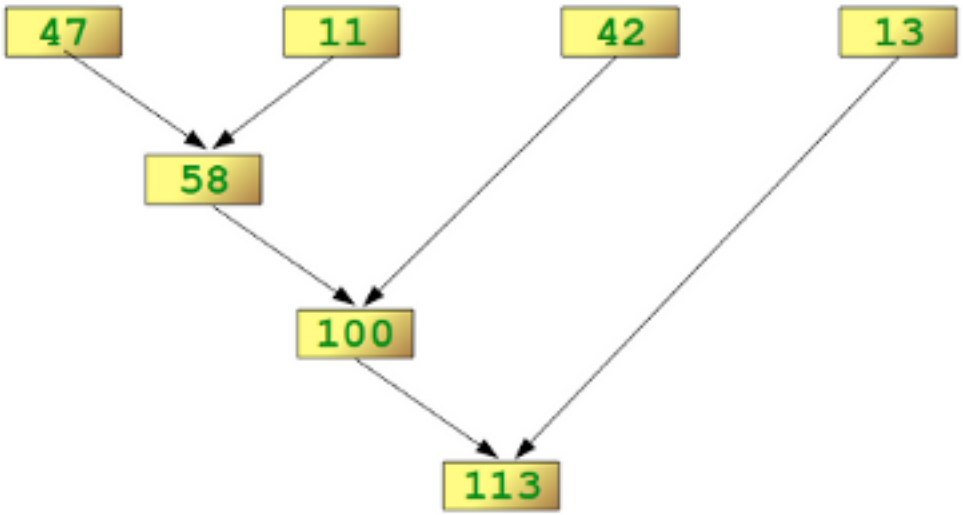
- Also like map, filter can be emulated by list comprehension syntax with often-simpler results

```
>>> [x for x in range(−5, 5) if x > 0]
[1, 2, 3, 4]
```

Functional Programming Tools

- The functional reduce call, which is a simple built-in function in 2.X but lives in the functools module in 3.X, is more complex.
- It accepts an iterable to process, but it's not an iterable itself—it returns a single result.
- The function reduce(func, seq) continually applies the function func() to the sequence seq. It returns a single value.
- Consider :

```
>>> import functools
>>>functools.reduce(lambda x,y : x+y, [47, 11, 42, 13])
113
```



Functional Programming Tools

- Here are two reduce calls that compute the sum and product of the items in a list:

```
>>> from functools import reduce # Import in 3.X, not in 2.X
>>> functools.reduce((lambda x, y: x + y), [1, 2, 3, 4])
10

>>> functools.reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

- At each step, reduce passes the current sum or product, along with the next item from the list, to the passed-in lambda function.
- By default, the first item in the sequence initializes the starting value.

Standard module - sys

module os (and sys and path)

- The os and sys modules provide numerous tools to deal with filenames, paths, directories.
- The os module contains two sub-modules os.sys (same as sys) and os.path that are dedicated to the system and directories; respectively.
- Whenever possible, you should use the functions provided by these modules for file, directory, and path manipulations.
- These modules are wrappers for platform-specific modules, so functions like **os.path.split** work on UNIX, Windows, Mac OS, and any other platform supported by Python.

```
>>> import os
>>> import os.path
>>> os.path.join(os.sep, 'home', 'user', 'work')
'/home/user/work'

>>> os.path.split('/usr/bin/python')
('/usr/bin', 'python')
```

Module os (and sys and path)

- The os module has lots of functions. Some of them are :
- The getcwd() function returns the current directory
- The chdir() – changes the current directory
- The listdir() – returns the content of a directory
- The mkdir() – creates a directory
- The rmdir() – removes a directory

- To remove a file : os.remove(filename)
- To rename a file : os.rename(oldfilename, newfilename)

- You can easily access all the environment variables : os.environ.keys()

Module os (and sys and path)

os.walk

- It generates the filenames in a directory tree by walking the tree either top-down or bottom-up.
- For each directory in the tree rooted at directory top (including top itself), it yields a 3-tuple(dirpath, dirnames, filename), where 'dirpath' -- is a string, the path to the directory; 'dirnames' – is a list of the names of the subdirectories in dirpath (excluding '.' and '..'); 'filenames' – is a list of the names of the non-directory files in dirpath.
- Please note :: that the names in the lists contain no path components. So to get a full path(which begins with top) to a file or directory in dirpath, you should use

os.path.join(dirpath, name)

Module os (and sys and path)

fnmatch

- The fnmatch module compares file names against glob-style patterns such as used by Unix shells. These are not the same as the more sophisticated regular expression rules. It's purely a string matching operation. If you find it more convenient to use a different pattern style, for example regular expressions, then simply use regex operations to match your filenames.
- The fnmatch module is used for the wild-card pattern matching.
- **Simple Matching** -- fnmatch() compares a single file name against a pattern and returns a boolean indicating whether or not they match.
- The comparison is case-sensitive when the operating system uses a case-sensitive file system.

- **Filtering** -- To test a sequence of filenames, you can use filter(). It returns a list of the names that match the pattern argument.

Module os (and sys and path)

- Let us consider two simple [but very useful] examples to demonstrate the usage

```
##  searches only .mp3 files
import fnmatch
import os

rootPath = '/'
pattern = '*.mp3'

for root, dirs, files in os.walk(rootPath):
        for filename in fnmatch.filter(files, pattern):
                print(os.path.join(root, filename))
```

Module os (and sys and path)

- Let us consider two simple [but very useful] examples to demonstrate the usage

```
##  searches only image files
import fnmatch
import os

rootPath = '/Users/prasun/'
images = ['*.jpg', '*.jpeg', '*.png', '*.tif', '*.tiff']
matches = []

for root, dirnames, filenames in os.walk(rootPath):
        for extensions in images:
                for filename in fnmatch.filter(filenames, extensions):
                        matches.append(os.path.join(root, filename))
print(matches)
```

Module os (and sys and path)

The sys module

- Like all the other modules, the sys module has to be imported with the import statement, i.e.

import sys

- The sys module provides information about constants, functions and methods of the Python interpreter.
- When starting a Python shell, Python provides 3 file objects called stadnard input, stadn output and standard error.
- There are accessible via the sys module:

sys.stderr

sys.stdin

sys.stdout

- The sys.argv is used to retrieve user argument when your module is executable.

Module os (and sys and path)

The sys module

```
>>> import sys
>>> sys.version
'3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09) \n[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)]'
```

Command-Line arguments

- Lots of scripts need access to the arguments passed to the script, when the script was started.
- sys.argv is a list, which contains the command-line arguments passed to the script.
- The first item of this list contains the name of the script itself. The arguments follow the script name.

- The following script iterates over the sys.argv list :

Module os (and sys and path)

The sys module

```
import sys
# it's easy to print this list of course:
print (sys.argv)
# or it can be iterated via a for loop:
for i in range(len(sys.argv)):
        if i == 0:
                print ("Function name: %s" % sys.argv[0] )
        else:
                print ("%d. argument: %s" % (i,sys.argv[i]) )
```

- We save this script as arguments.py. If we call it, we get the following output::

```
$ python arguments.py arg1 arg2
['arguments.py', 'arg1', 'arg2']
Function name: arguments.py
1. argument: arg1
2. argument: arg2
$
```

Module os (and sys and path)

The sys module

- Every serious user of a UNIX or Linux operating system knows standard streams, i.e. input, standard output and standard error.
- They are known as pipes.
- They are commonly abbreviated as stdin, stdout, stderr.
- The standard input (stdin) is normally connected to the keyboard, while the standard error and standard output go to the terminal (or window) in which you are working.
- These data streams can be accessed from Python via the objects of the sys module with the same names, i.e. sys.stdin, sys.stdout and sys.stderr.

```
>>> import sys
>>> for I in (sys.stdin, sys.stdout, sys.stderr):
....        print(i)
....
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
```

Module os (and sys and path)

The sys module

- Consider the following example:

```
import sys
while True:
    # output to stdout:
    print ("Yet another iteration ..." )
    try:
        # reading from sys.stdin (stop with Ctrl-D):
        number = input("Enter a number: ")
    except EOFError:
        print ("\nciao" )
        break
    else:
        number = int(number)
        if number == 0:
            print ("0 has no inverse”)
        else:
            print ("inverse of %d is %f" % (number, 1.0/number))
```

Module os (and sys and path)

The sys module

- If we save the previous example under "streams.py" and use a file called "number.txt" with numbers (one number per line) for the input, we can call the script in the following way from the Bash shell:

$ python streams.py < numbers.txt

- It's also possible to redirect the output into a file:

$ python streams.py < numbers.txt > output.txt

Standard module - math

- To use the functions of this module, you need to import it

import math

- Some of the functions are :

math.ceil(x) – return the ceiling of x, the smallest integer >= to x.

math.fabs(*x*) -- Return the absolute value of *x*.

math.factorial(*x*) -- Return *x* factorial.

math.floor(*x*) -- Return the floor of *x*, the largest integer <= x .

math.fsum(*iterable*) -- Return an accurate floating point sum of values in the iterable.

math.modf(*x*) -- Return the fractional and integer parts of *x*.

math.trunc(*x*) -- Return the Real value *x* truncated to an Integral (usually an integer).

Standard module - math

- Some of the functions are [contd.] :

math.exp(*x*) -- Return e**x.

math.log(*x*[, *base*]) -- Return the logarithm of *x* to the given *base*. If the *base* is not specified, return the natural logarithm of *x* (that is, the logarithm to base *e*).

math.log10(*x*) -- Return the base-10 logarithm of *x*.

math.pow(*x*, *y*) -- Return x raised to the power y.

math.sqrt(*x*) -- Return the square root of *x*.

All trigonometric functions like :

math.acos(*x*) -- Return the arc cosine of *x*, in radians.

math.cos(*x*) -- Return the cosine of *x* radians.

math.sin(*x*) -- Return the sine of *x* radians.

math.tan(*x*) -- Return the tangent of *x* radians.

Standard module - math

- Some of the functions are [contd.] :

All hyperbolic functions like math.acosh(x)

math.pi – The mathematical constant pi

math.e -- The mathematical constant e

Note : The math module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases is loosely specified by the C standards, and Python inherits much of its math-function error-reporting behavior from the platform C implementation. As a result, the specific exceptions raised in error cases (and even whether some arguments are considered to be exceptional at all) are not defined in any useful cross-platform or cross-release way.

Standard module - time

- This module provides various time-related functions.
- Although this module is always available, not all functions are available on all platforms.
- Most of the functions defined in this module call platform C library functions with the same name.
- The *epoch* is the point where the time starts.
- On January 1st of that year, at 0 hours, the “time since the epoch” is zero.
- For Unix, the epoch is 1970.
- To find out what the epoch is, look at gmtime(0).
- The functions in this module do not handle dates and times before the epoch or far in the future.
- The cut-off point in the future is determined by the C library; for Unix, it is typically in 2038.

Standard module - time

- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT).
- The acronym UTC is not a mistake but a compromise between English and French.
- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year.
- DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.
- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock "ticks" only 50 or 100 times a second.

Standard module - time

- Many of Python's time functions handle time as a tuple of 9 numbers, as shown below

Index	Field	Values
0	4-digit year	2008
1	Month	1 to 12
2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 61 (60 or 61 are leap-seconds)
6	Day of Week	0 to 6 (0 is Monday)
7	Day of year	1 to 366 (Julian day)
8	Daylight savings	-1, 0, 1, -1 means library determines DST

Standard module - time

- The above tuple is equivalent to **struct_time** structure. This structure has following attributes

Index	Attributes	Values
0	tm_year	2008
1	tm_mon	1 to 12
2	tm_mday	1 to 31
3	tm_hour	0 to 23
4	tm_min	0 to 59
5	tm_sec	0 to 61 (60 or 61 are leap-seconds)
6	tm_wday	0 to 6 (0 is Monday)
7	tm_yday	1 to 366 (Julian day)
8	tm_isdst	-1, 0, 1, -1 means library determines DST

Standard module - time

- Getting formatted current time:

```
import time;
localtime = time.asctime(time.localtime(time.time()))
print ("Local current time :", localtime)
```

- This would produce the following result :

```
Local current time :  Tue Jun 28 09:56:14 2016
```

- The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.
- By default, calendar takes Monday as the first day of the week and Sunday as the last one.

- There are several functions, some of them are :

time.sleep(*secs*) -- Suspend execution for the given number of seconds.

time.time() -- Return the time as a floating point number expressed in seconds since the epoch, in UTC.

module - datetime

- The datetime module supplies classes for manipulating dates and times in both simple and complex ways.
- While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation.
- Date arithmetic

```
>>> import datetime
>>> today = datetime.date.today()
>>> print ('Today :', today)
Today : 2016-06-28
>>> one_day = datetime.timedelta(days=1)
>>> print ('One day :', one_day)
One day : 1 day, 0:00:00
>>> yesterday = today - one_day
>>> print ('Yesterday:', yesterday)
Yesterday: 2016-06-27
>>> tomorrow = today + one_day
>>> print ('Tomorrow :', tomorrow)
Tomorrow : 2016-06-29
```

module - datetime

- An example :

```
import time
import datetime
print  ("Time in seconds since the epoch: %s" %time.time() )
print  ("Current date and time: " , datetime.datetime.now() )
print  ("Or like this: " ,datetime.datetime.now().strftime("%y-%m-%d-%H-%M") )
print  ("Current year: ", datetime.date.today().strftime("%Y") )
print  ("Month of year: ", datetime.date.today().strftime("%B") )
print  ("Week number of the year: ", datetime.date.today().strftime("%W") )
print ("Weekday of the week: ", datetime.date.today().strftime("%w") )
print ("Day of year: ", datetime.date.today().strftime("%j") )
print ("Day of the month : ", datetime.date.today().strftime("%d") )
print  ("Day of week: ", datetime.date.today().strftime("%A"))
```

module - datetime

- An example :

The output will be something like below :

Time in seconds since the epoch: 1467089053.840859
Current date and time: 2016-06-28 10:14:13.840894
Or like this: 16-06-28-10-14
Current year: 2016
Month of year: June
Week number of the year: 26
Weekday of the week: 2
Day of year: 180
Day of the month : 28
Day of week: Tuesday

module - datetime

- Another very commonly used example :

```
from datetime import datetime

noofelements = 10000

starttime = datetime.now() # record starttime
# do anything here − below
pnlist1 = [x + y for x in range(noofelements) for y in range(noofelements)]

time1 = datetime.now() - starttime # record the time difference

print('No of initial elements : ', noofelements)
print("Length of list1 : ", len(pnlist1))
# print the time difference
print('List Comprehension takes : ', time1, "seconds")
```

Reference

- Python for Informatics – C. Severance
- Think Python – A. B. Downey - [O'Reilly]
- Python Crash Course – Eric Matthes [No starch Press]
- A Byte of Python – Swaroop C H
- Introducing Python – Bill Lubanovic [O'Reilly]

- Learning Python – 5th Ed – Mark Lutz - [O'Reilly]

End of Presentation

Python – S6