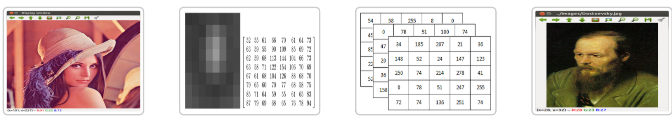
Python Functions - lambda 2016



List of Python Tutorials

Functions lambda

Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called **lambda**. This is not exactly the same as lambda in functional programming languages such as Lisp, but it is a very powerful concept that's well integrated into Python and is often used in conjunction with typical functional concepts like **filter()**, **map()** and **reduce()**.

Like **def**, the **lambda** creates a function to be called later. But it returns the function instead of assigning it to a name. This is why **lambda**s are sometimes known as **anonymous** functions. In practice, they are used as a way to inline a function definition, or to defer execution of a code.

The following code shows the difference between a normal function definition, **func** and a lambda function, **lamb**:

```
>>>
>>> def func(x): return x ** 3

>>> print(func(5))
125
>>>
>>> lamb = lambda x: x ** 3
>>> print(lamb(5))
125
>>>
```

As we can see, func() and lamb() do exactly the same and can be used in the same ways. Note that the lambda definition does not include a **return** statement -- it always contains an expression which is returned. Also note that we can put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all.

The **lambda**'s general form is :

```
lambda arg1, arg2, ...argN : expression using arguments
```

Function objects returned by running **lambda** expressions work exactly the same as those created and assigned by **def**s. However, there are a few differences that make **lambda** 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. For example, places like inside a list literal, or a function call's arguments. As

Sets (union/intersection) and itertools - Jaccard coefficient & shingling to check plagiarism
Hashing (Hash tables & hashlib)
Dictionary Comprehension with zip
The yield keyword
Generator Functions and Expressions
generator.send() method
Iterators
Iterators II
Classes and Instances (__init__, __call__, etc.)
if__name__ == '__main__'
argparse
@static method vs class method
Private attributes and private methods
bits, bytes, bitstring, and constBitStream
Python Object Serialization - pickle and json
Python Object Serialization - yaml and json
Priority queue and heap queue data structure
Graph data structure
Dijkstra's shortest path algorithm
Prim's spanning tree algorithm
Closure
Functional programming in Python
Remote running a local file using ssh
SQLite 3 - A. Connecting to DB, create/drop table, and insert data into a table
SQLite 3 - B. Selecting, updating and deleting data
MongoDB with PyMongo I - Installing MongoDB ...

an expression, **lambda** returns a value 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 is as a result.

- **lambda's body is a single expression, not a block of statements**. The **lambda**'s body is similar to what we'd put in a **def** body's **return** statement. We simply type the result as an expression instead of explicitly returning it. Because it is limited to an expression, a **lambda** is less general that a **def**. We can only squeeze design, to limit program nesting. **lambda** is designed for coding simple functions, and **def** handles larger tasks.

```
>>>
>>> def f(x, y, z): return x + y + z

>>> f(2, 30, 400)
432
```

We can achieve the same effect with **lambda** expression by explicitly assigning its result to a name through which we can call the function later:

```
>>>
>>> f = lambda x, y, z: x + y + z
>>> f(2, 30, 400)
432
>>>
```

Here, **f** is assigned the function object the **lambda** expression creates. This is how **def** works, too. But in **def**, its assignment is an automatic must.

Default work on **lambda** arguments:

```
>>> mz = (lambda a = 'Wolfgangus', b = ' Theophilus', c = ' Mozart
>>> mz('Wolfgang', ' Amadeus')
'Wolfgang Amadeus Mozart'
>>>
```

In the following example, the value for the name **title** would have been passes in as a default argument value:

```
>>> def writer():
        title = 'Sir'
        name = (lambda x:title + ' ' + x)
        return name

>>> who = writer()
>>> who('Arthur Ignatius Conan Doyle')
'Sir Arthur Ignatius Conan Doyle'
>>>
```

Why lambda?

The **lambda**s can be used as a function shorthand that allows us to embed a function within the code. For instance, callback handlers are frequently coded as inline **lambda** expressions embedded directly in a registration call's arguments list. Instead of being define with a **def** elsewhere in a file and referenced by name, **lambda**s are also commonly used to code **jump tables** which are lists or dictionaries of actions to be performed on demand.

```
>>>
>>> L = [lambda x: x ** 2,
         lambda x: x ** 3,
         lambda x: x ** 4]
```

Python HTTP Web Services - urllib, httplib2
Web scraping with Selenium for checking domain availability
Multithreading ...
Python Network Programming I - Basic Server / Client : A Basics
Python Network Programming I - Basic Server / Client : B File Transfer
Python Network Programming II - Chat Server / Client
Python Network Programming III - Echo Server using socketserver network framework
Python Network Programming IV - Asynchronous Request Handling : ThreadingMixIn and ForkingMixIn
Python Interview Questions I
Python Interview Questions II
Python Interview Questions III
Python & C++ with SIP
PyDev with Eclipse
Matplotlib
NumPy array basics A
NumPy Matrix and Linear Algebra
Celluar Automata
Batch gradient descent algorithm
Longest Common Substring Algorithm
Python Unit Test - TDD using unittest.TestCase class
Simple tool - Google page ranking by keywords
Google App Hello World
Google App webapp2 & WSGI
Uploading Google App Hello World
Python 2 vs Python 3

```
>>> for f in L:
        print(f(3))


9
27
81
>>> print(L[0](11))
121
>>>
```

In the example above, a list of three functions was built up by embedding **lambda** expressions inside a list. A **def** won't work inside a list literal like this because it is a statement, not an expression. If we really want to use **def** for the same result, we need temporary function names and definitions outside:

```
>>>
>>> def f1(x): return x ** 2

>>> def f2(x): return x ** 3

>>> def f3(x): return x ** 4

>>> # Reference by name
>>> L = [f1, f2, f3]
>>> for f in L:
        print(f(3))


9
27
81
>>> print(L[0](3))
9
>>>
```

We can use dictionaries doing the same thing:

>>> key = 'quadratic' >>> {'square': (lambda x: x ** 2), 'cubic': (lambda x: x ** 3), 'quadratic': (lambda x: x ** 4)}[key](10) 10000 >>>

Here, we made the temporary dictionary, each of the nested **lambda**s generates and leaves behind a function to be called later. We fetched one of those functions by indexing and the parentheses forced the fetched function to be called.

Again, let's do the same thing without **lambda**.

```
>>>
>>> def f1(x): return x ** 2

>>> def f2(x): return x ** 3

>>> def f3(x): return x ** 4

>>> key = 'quadratic'
>>> {'square': f1, 'cubic': f2, 'quadratic': f3}[key](10)
10000
>>>
```

This works but our **def**s may be far away in our file. The **code proximity** that **lambda** provide is useful for functions that will only be used in a single context. Especially, if the three functions are not going to be used anywhere else, it makes sense to embed them within the dictionary as **lambda**s. Also, the **def** requires more names for these title functions that may cause name clash with other names in this file.

If we know what we're doing, we can code most statements as expressions:

```
>>>
>>> min = (lambda x, y: x if x < y else y)
```


virtualenv and virtualenvwrapper

Uploading a big file to AWS S3 using boto module

Scheduled stopping and starting an AWS instance

Cloudera CDH5 - Scheduled stopping and starting services

Removing Cloud Files - Rackspace API with curl and subprocess

Checking if a process is running/hanging and stop/run a scheduled task on Windows

Apache Spark 1.3 with PySpark (Spark Python API) Shell

Apache Spark 1.2 Streaming

bottle 0.12.7 - Fast and simple WSGI-micro framework for small web-applications ...

Fabric - streamlining the use of SSH for application deployment

Neural Networks with backpropagation for XOR using one hidden layer

NLP - NLTK (Natural Language Toolkit) ...

RabbitMQ(Message broker server) & Celery(Task queue) ...

OpenCV3 & Matplotlib ...

Simple tool - Concatenating slides using FFmpeg ...

iPython - Signal Processing with NumPy

Downloading YouTube videos using youtube-dl embedded with Python

Machine Learning : scikit-learn ...

Django 1.6/1.8 Web Framework ...

```
>>> min(101*99, 102*98)
9996
>>> min(102*98, 101*99)
9996
>>>
```

If we need to perform loops within a **lambda**, we can also embed things like **map** calls and list comprehension expressions.

```
>>> import sys
>>> fullname = lambda x: list(map(sys.stdout.write,x))
>>> f = fullname(['Wassily ', 'Wassilyevich ', 'Kandinsky'])
Wassily Wassilyevich Kandinsky
>>>
>>>
>>> fullname = lambda x: [sys.stdout.write(a) for a in x]
>>> t = fullname(['Wassily ', 'Wassilyevich ', 'Kandinsky'])
Wassily Wassilyevich Kandinsky
>>>
```

Here is the description of **map** built-in function.

map(function, iterable, ...)
Return an iterator that applies **function** to every item of **iterable**, yielding the results. If additional **iterable** arguments are passed, **function** must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted.

So, in the above example, **sys.stdout.write** is an argument for **function**, and the **x** is an iterable item, list, in the example.

Nested lambda

In the following example, the **lambda** appears inside a **def** and so can access the value that the name **x** has in the function's scope at the time that the enclosing function was called:

```
>>> def action(x):
        # Make and return function, remember x
        return (lambda newx: x + newx)

>>> ans = action(99)
>>> ans
<function <lambda> at 0x0000000003334648>
>>> ans(100)
199
>>>
```

Though not clear in this example, note that **lambda** also has access to the names in any enclosing **lambda**. Let's look at the following example:

```
>>>
>>> action = (lambda x: (lambda newx: x + newx))
>>> ans = action(99)
>>> ans
<function <lambda> at 0x0000000003308048>
>>> ans(100)
199
>>>
>>> (  (lambda x: (lambda newx: x + newx)) (99)) (100)
199
```

In the example, we nested **lambda** structure to make a function that makes a

- K Hong

OpenCV 3 image & video processing with Python

OpenCV 3 with Python
Image - OpenCV BGR : Matplotlib RGB
Basic image operations - pixel access
iPython - Signal Processing with NumPy
Signal Processing with NumPy I - FFT & DFT for sine, square waves, unitpulse, and random signal
Signal Processing with NumPy II - Image Fourier Transform : FFT & DFT
Inverse Fourier Transform of an Image with low pass filter: cv2.idft()
Image Histogram
Video Capture & Switching colorspaces - RGB / HSV
Adaptive Thresholding - Otsu's clustering-based image thresholding
Edge Detection - Sobel

function when called. It's fairly convoluted and it should be avoided.

lambda and sorted()

Here is a simple example of using **lambda** with built-in function **sorted()**:

```
sorted(iterable[, key][, reverse])
```

The **sorted()** have a key parameter to specify a function to be called on each list element prior to making comparisons.

```
>>> death = [
    ('James', 'Dean', 24),
    ('Jimi', 'Hendrix', 27),
    ('George', 'Gershwin', 38),
]
>>> sorted(death, key=lambda age: age[2])
[('James', 'Dean', 24), ('Jimi', 'Hendrix', 27), ('George', 'Gersh
```

In this example, we want to read a video file and sort the packet in the order of starting time stamp. Also, we want to count the number of chunks.

```
#!/usr/bin/python
import psutil
import simplejson
import subprocess

procs_id = 0
procs = {}
procs_data = []

def getMetadata(video):
    cmd = ['ffprobe', '-show_streams', '-show_packets', '-print_fc
    print 'cmd=', cmd
    stdout = runCommand(cmd, return_stdout = True, busy_wait = Fal
    data = simplejson.loads(stdout)
    metadata = { }

    if data:
        # Obtain duration here
        if 'streams' in data:
            for item in data['streams']:
                if 'codec_type' in item and 'duration' in item and
                    metadata['duration'] = float(item['duration'])
        else:
            metadata['duration'] = float(0)

        # Obtain iframes here
        iframes = []
        if 'packets' in data:
            # Filter out packet types
            video_packets = sorted(
                [packet for packet in data['packets'] if (packet['
                key = lambda packet: int(packet['pos'])
            )
            video_positions = sorted([int(packet['pos']) for packe
            audio_packets = sorted(
                [packet for packet in data['packets'] if (packet['
                key = lambda packet: int(packet['pos']))
            audio_positions = sorted([int(packet['pos']) for packe

            # Search for iframes
            iframe_packets = [packet for packet in video_packets i
            positions = sorted([int(packet['pos']) for packet in d

            start_byte = 0
            end_byte = 0
            duration = None

            for iframe in iframe_packets:
                start_byte = int(iframe['pos'])
```


Machine Learning with scikit-learn

```
                end_byte = 0

                for pos in positions:
                    if pos > start_byte:
                        end_byte = pos - 188
                        break

                if duration is None:
                    duration = float(iframe['pts_time'])
                else:
                    new_duration = float(iframe['pts_time'])
                    iframes.append({ 'byte_start': start_byte,
                                     'byte_end': end_byte,
                                     'duration': (new_duration - d
                    duration = new_duration

            last_duration = float(video_packets[-1]['pts_time'])
            iframes.append({ 'byte_start': start_byte,
                             'byte_end': end_byte,
                             'duration': last_duration - duration
        metadata['iframes'] = iframes
        print 'metadata=',metadata

    return metadata


# Runs command silently
def runCommand(cmd, use_shell = False, return_stdout = False, busy
    # Sanitize cmd to string
    cmd = map(lambda x: '%s' % x, cmd)
    if use_shell:
        command = ' '.join(cmd)
    else:
        command = cmd

    if return_stdout:
        proc = psutil.Popen(cmd, shell = use_shell, stdout = subpr
    else:
        proc = psutil.Popen(cmd, shell = use_shell,
                                stdout = open('/dev/null', 'w'),
                                stderr = open('/dev/null', 'w'))


    global procs_id
    global procs
    global procs_data
    proc_id = procs_id
    procs[proc_id] = proc
    procs_id += 1
    data = { }

    while busy_wait:
        returncode = proc.poll()
        if returncode == None:
            try:
                data = proc.as_dict(attrs = ['get_io_counters', 'g
            except Exception, e:
                pass
            time.sleep(poll_duration)
        else:
            break

    (stdout, stderr) = proc.communicate()
    returncode = proc.returncode
    del procs[proc_id]

    if returncode != 0:
        raise Exception(stderr)
    else:
        if data:
            procs_data.append(data)
        return stdout

if __name__ == '__main__':

    segMeta = getMetadata('bunny_400.ismv')
    print 'segMeta=',segMeta
    for k in segMeta.keys():
        if(k == 'iframes'):
            print 'iframe size =',len(segMeta[k])
            break
```

After reading in the **video** using **ffprobe**, the **data** looks like this:

```
{
    "packets": [
        {
```

```
            "codec_type": "video",
            "stream_index": 0,
            "pts": 0,
            "pts_time": "0.000000",
            "dts": 0,
            "dts_time": "0.000000",
            "size": "847",
            "pos": "2927",
            "flags": "K"
        },
        {
            "codec_type": "video",
            "stream_index": 0,
            "pts": 1200000,
            "pts_time": "0.120000",
            "dts": 1200000,
            "dts_time": "0.120000",
            "size": "486",
            "pos": "3804",
            "flags": "_"
        },
        ........
    ],
    "streams": [
        {
            "index": 0,
            "codec_name": "h264",
            "codec_long_name": "H.264 / AVC / MPEG-4 AVC / MPEG-4
            "profile": "High",
            "codec_type": "video",
            "codec_time_base": "1/50",
            "codec_tag_string": "avc1",
            "codec_tag": "0x31637661",
            "width": 288,
            "height": 160,
            "has_b_frames": 2,
            "sample_aspect_ratio": "80:81",
            "display_aspect_ratio": "16:9",
            "pix_fmt": "yuv420p",
            "level": 13,
            "r_frame_rate": "25/1",
            "avg_frame_rate": "0/0",
            "time_base": "1/10000000",
            "start_pts": 0,
            "start_time": "0.000000",
            "duration_ts": 5964400000,
            "duration": "596.440000",
            "bit_rate": "400074",
            "nb_read_packets": "14911",
            "disposition": {
                "default": 1,
                "dub": 0,
                "original": 0,
                "comment": 0,
                "lyrics": 0,
                "karaoke": 0,
                "forced": 0,
                "hearing_impaired": 0,
                "visual_impaired": 0,
                "clean_effects": 0,
                "attached_pic": 0
            },
            "tags": {
                "language": "und",
                "handler_name": "VideoHandler"
            }
        }
    ]
}
```


The input file is: video.dat which is actually a fragmented mp4 file.

Output looks like this:

```
cmd= ['ffprobe', '-show_streams', '-show_packets', '-print_format'
metadata= {
'duration': 596.44,
'iframes': [
{'duration': 10.0, 'byte_end': 399823, 'byte_start': 377082},
{'duration': 10.0, 'byte_end': 998254, 'byte_start': 984197},
{'duration': 10.0, 'byte_end': 1833216, 'byte_start': 1804498},
{'duration': 10.0, 'byte_end': 2591816, 'byte_start': 2569925},
....
{'duration': 10.0, 'byte_end': 29431348, 'byte_start': 29422617},
{'duration': 10.0, 'byte_end': 29633871, 'byte_start': 29633940},
{'duration': 10.0, 'byte_end': 29801180, 'byte_start': 29793525},
```

```
{'duration': 6.399999999999977, 'byte_end': 29801180, 'byte_start'
iframe size = 60
```

List of Python Tutorials

- Graph data structure
- Dijkstra's shortest path algorithm
- Prim's spanning tree algorithm
- Closure
- Functional programming in Python
- Remote running a local file using ssh
- SQLite 3 - A. Connecting to DB, create/drop table, and insert data into a table
- SQLite 3 - B. Selecting, updating and deleting data
- MongoDB with PyMongo I - Installing MongoDB ...
- Python HTTP Web Services - urllib, httplib2
- Web scraping with Selenium for checking domain availability
- Multithreading ...
- Python Network Programming I - Basic Server / Client : A Basics
- Python Network Programming I - Basic Server / Client : B File Transfer
- Python Network Programming II - Chat Server / Client
- Python Network Programming III - Echo Server using socketserver network framework
- Python Network Programming IV - Asynchronous Request Handling : ThreadingMixIn and ForkingMixIn
- Python Interview Questions I
- Python Interview Questions II
- Python Interview Questions III
- Python & C++ with SIP
- PyDev with Eclipse
- Matplotlib
- NumPy array basics A
- NumPy Matrix and Linear Algebra
- Celluar Automata
- Batch gradient descent algorithm
- Longest Common Substring Algorithm
- Python Unit Test - TDD using unittest.TestCase class
- Simple tool - Google page ranking by keywords
- Google App Hello World
- Google App webapp2 & WSGI
- Uploading Google App Hello World
- Python 2 vs Python 3
- virtualenv and virtualenvwrapper
- Uploading a big file to AWS S3 using boto module
- Scheduled stopping and starting an AWS instance
- Cloudera CDH5 - Scheduled stopping and starting services
- Removing Cloud Files - Rackspace API with curl and subprocess
- Checking if a process is running/hanging and stop/run a scheduled task on Windows
- Apache Spark 1.3 with PySpark (Spark Python API) Shell
- Apache Spark 1.2 Streaming
- bottle 0.12.7 - Fast and simple WSGI-micro framework for small web-applications ...
- Fabric - streamlining the use of SSH for application deployment

- Neural Networks with backpropagation for XOR using one hidden layer
- NLP - NLTK (Natural Language Toolkit) ...
- RabbitMQ(Message broker server) & Celery(Task queue) ...
- OpenCV3 & Matplotlib ...
- Simple tool - Concatenating slides using FFmpeg ...
- iPython - Signal Processing with NumPy
- Downloading YouTube videos using youtube-dl embedded with Python
- Machine Learning : scikit-learn ...
- Django 1.6/1.8 Web Framework ...