CONTENTS

- Python 2.x vs. 3.x
- Branching, Looping, & Exceptions
- The Zen of Python
- Python Operators
- Instantiating Classes...and more!

Core Python

Creating Beautiful Code with an Interpreted, Dynamically Typed Language

UPDATED BY IVAN MUSHKETYK
ORIGINAL BY NAOMI CEDER AND MIKE DRISCOLL

INTRODUCTION

Python is an interpreted, dynamically typed language. Python uses indentation to create readable, even beautiful, code. Python comes with so many libraries that you can handle many jobs with no further libraries. Python fits in your head and tries not to surprise you, which means you can write useful code almost immediately.

Python was created in 1990 by Guido van Rossum. While the snake is used as a mascot for the language and community, the name is actually derived from Monty Python, and references to Monty Python skits are common in code examples and library names. There are several other popular implementations of Python, including PyPy (JIT compiler), Jython (JVM integration), and IronPython (.NET CLR integration).

PYTHON 2.X VS. PYTHON 3.X

Python comes in two basic flavors these days—Python 2.x (currently 2.7) and Python 3.x (currently 3.3). This is an important difference—some code written for one won't run on the other. However, most code is interchangeable. Here are some of the key differences:

Python 2.x	Python 3.x
`print "hello"` (`print` is a keyword)	`print("hello")` (`print` is a function)
except Exception, e: # OR except Exception as e	except Exception as e: # ONLY
Naming of Libraries and APIs are frequently inconsistent with PEP 8	Improved (but still imperfect) consistency with PEP 8 guidelines
Strings **and** unicode	Strings are **all** unicode and bytes type is for unencoded 8 bit values

There is a utility called 2to3.py that you can use to convert Python 2.x code to 3.x, while the '-3' command line switch in 2.x enables additional deprecation warnings for cases the automated converter cannot handle. Third party tools like python-modernize and the 'six' support package make it easy to target the large common subset of the two variants for libraries and applications which support both 2.x and 3.x.

LANGUAGE FEATURES

PROGRAMMING AS GUIDO INTENDED IT...

Indentation rules in Python. There are no curly braces, no begin or end keywords, no need for semicolons at the ends of lines—the only thing that organizes code into blocks, functions, or classes is indentation. If something is indented, it forms a block with everything indented at the same level until the end of the file or the next line with less indentation.

While there are several options for indentation, the common standard is 4 spaces per level:

```
def function_block():
    # first block stuff
    # second block within first block:
        # second block stuff
        for x in an_iterator:
            # this is the block for the for loop
            print x
        # back out to this level ends the for loop
    # and this ends the second block...
    # more first block stuff
def another_function_block():
```

COMMENTS AND DOCSTRINGS

To mark a comment from the current location to the end of the line, use a pound sign, '#'.

```
# this is a comment on a line by itself
x = 3    # this is a partial line comment after some code
```

For longer comments and more complete documentation, especially at the beginning of a module or of a function or class, use a triple quoted string. You can use 3 single or double quotes. Triple quoted strings can cover multiple lines and any unassigned string in a Python program is ignored. Such strings are often used for documentation of modules, functions, classes, and methods. By convention, the "docstring" is the first statement in its enclosing scope. Following this convention allows automated production of documentation using the pydoc module.

In general, you use one-line comments for commenting code from the point of view of a developer trying to understand the code itself. Docstrings are more properly used to document what the code does, more from the point of view of someone who is going to be using the code.

Python is the sort of language that you can just dive into, so let's start with this example Python script:

INTRODUCING THE

Security Zone

Arm Yourselves From Hacks and Data Leaks!

Retroactively injecting security into applications is unnecessary and expensive. To avoid these costly patches, secure development has become essential for preventing hacks.

Our newest Zone addresses security issues head-on, offering tools, tutorials, and updates to secure your applications and the machines that run them!

dzone.com/security

```
#! /usr/bin/env python
""" An example Python script
    Note that triple quotes allow multiline strings
"""

# single line comments are indicated with a "#"

import sys         # loads the sys (system) library

def main_function(parameter):
    """ This is the docstring for the function """
    print "here is where we do stuff with the parameter"
    print parameter

    return a_result     # this could also be multiples

if __name__ == "__main__":
 """ this will only be true if the script is called
        as the main program """
    # command line parameters are numbered from 0
    # sys.argv[0] is the script name
    param = sys.argv[1]    # first param after script name
    # the line below calls the main_function and
    # puts the result into function_result
    function_result = main_function(param)
```

BRANCHING, LOOPING, AND EXCEPTIONS

BRANCHING

Python has a very straightforward set of if/else statements:

```
if something_is_true:
    # do this
elif something_else_is_true:
    # do that
else:
    # do the other thing
```

The expressions that are part of if and elif statements can be comparisons (==, <, >, <=, >=, etc) or they can be any python object. In general, zero and empty sequences are False, and everything else is True. Python does not have a switch statement.

LOOPS

Python has two loops. The for loop iterates over a sequence, such as a list, a file, or some other series:

```
for item in ['spam', 'spam', 'spam', 'spam']:
    print item
```

The code above will print "spam" four times. The while loop executes while a condition is true:

```
counter = 5
while counter > 0:
     counter -= 1
```

With each iteration, the counter variable is reduced by one. This code executes until the expression is False, which in this case is when `counter` reaches zero.

A better way to iterate over consecutive numbers is use the `range` function that returns an iterator from 0 to n-1:

```
for i in range(5):
    print i # Prints numbers from 0 to 4
```

The `range` function supports other modes that allow to specify the start value and the step value:

```
range(4, 10) # Returns values: 4, 5, 6, 7, 8, 9
range(4, 10, 2) # Returns values: 4, 6, 8
```

HANDLING EXCEPTIONS

Python is different from languages like C or Java in how it thinks about errors. Languages like Java are “look before you leap” (LBYL) languages. That is, there is a tendency to check types and values to make sure that they are legal before they are used. Python, on the other hand, thinks of things more in a “easier to ask for forgiveness than permission” (EAFP) style. In other words, Python’s style is more likely to go ahead and try an operation and then handle any problems if they occur:

```
try:
    item = x[0]
except TypeError:
    #this will print only on a TypeError exception
    print "x isn't a list!"
else:
    # executes if the code in the "try" does NOT
    # raise an exception
    print "You didn't raise an exception!"
finally:
    #this will always print
    print "processing complete"
```

In this case, a list or sequence operation is attempted, and if it fails because it's the wrong type, the except clause just deals with it. Otherwise the exception will be raised normally. Then, whether an exception happens or not, the finally clause will be executed, usually to clean up after the operation in either case.

Keyword	Usage
if <expression>:	Conditional expression that only executes if True
else:	Used primarily as a catchall. If <expression> is False, then we fall into the else
elif:	Use elif to test multiple conditions.
while <expression>:	The while loop only loops while an expression evaluates to True.
break	Breaks out of a loop
continue	Ends current iteration of loop and goes back to top of loop
try:	Begins a block to check for exceptions
except <exception>:	Followed by Exception type being checked for, begins block of code to handle exception
finally	Code that will be executed whether exception occurs or not

DATA OBJECTS

VARIABLES AND TYPES

Python is a dynamically typed language, but it is also a strongly typed language. So, a variable could end up referring to different types of objects, but the object that it's referring to at any given moment is strongly typed. For example:

```
x = 1      # x points to an integer object
y = 2      # y also points to an integer object
z = x + y  # z points to an integer object – 3
a = y      # a points to the same int object as y
y = "2"    # y now points to a different object, a string
z = x + y  # throws a type mismatch (TypeError) exception
               # since an integer and a string are different
               # types and can't be added.
z = x + a  # z now points to an int (3), since a is
               # pointing to an int
```

Python 3.6 allows you to write underscores in numeric intervals to represent them in a more readable fashion:

```
i = 10_000_000_000
f = 5_123_456.789_123
```

DUCK TYPING—IF IT QUACKS LIKE A ...

While Python objects themselves are strongly typed, there is a large amount of flexibility in how they are used. In many languages, there is a pattern of checking your code to be sure an object is of the correct type before attempting an operation. This approach limits flexibility and code reuse—even slightly different objects (say, a tuple vs. a list) will require different explicit checking.

In Python, things are different. Because the exception handling is strong, we can just go ahead and try an operation. If the object we are operating on has the methods or data members we need, the operation succeeds. If not, the operation raises an exception. In other words, in the Python world, if something walks like a duck and quacks like a duck, we can treat it like a duck. This is called "duck typing".

PYTHON DATA TYPES

Python has several data types. The most commonly found ones are shown in the following table.

Type	Description
`int`	An integer of the same size as a long in C on the current platform.
`long`	An integer of unlimited precision (In Python 3.x this becomes an int).
`float`	A floating point number , usually a double in C on the current platform.
`complex`	Complex numbers have a real and an imaginary component, each is a float.
`bool`	True or False.

PYTHON BUILT-IN OBJECT TYPES

Python also has built-in object types that are closely related to the data types mentioned above. Once you are familiar with these two sets of tables, you will know how to code almost anything!

Type	Description
`list`	Mutable sequence, always in square brackets: [1, 2, 3]
`tuple`	Immutable sequence, always in parentheses: (a, b, c)
`dict`	Dictionary - key, value storage. Uses curly braces: {key:value}
`set`	Collection of unique elements unordered, no duplicates
`str`	String - sequence of characters, immutable
`unicode`	Sequence of Unicode encoded characters

PYTHON OPERATORS

The following table lists Python's common operators:

Operator	Action	Example
`+`	Adds items together; for strings and sequences concatenates	`1 + 1 -> 2` `"one" + "one" -> "oneone"`
`-`	subtraction	`1 - 1 -> 0`
`*`	multiplication, with strings, repeats string	`2 * 3 -> 6` `"one" * 2 -> "oneone"`

`/ (//)`	division, division of integers results in an integer with truncation in Python 2.x, a float in Python 3.x (// is integer division in Python 3.x)	`3/4 -> 0 (2.x)`
		`3/4 -> 0.75 (3.x)`
		`3//4 -> 0 (3.x)`

`**`	Exponent - raises a number to the given exponent	

SEQUENCE INDEXES AND SLICING

There are several Python types that are all sequential collections of items that you access by using numeric indexes, like lists, tuples, and strings. Accessing a single item from one of these sequences is straightforward—just use the index, or a negative index to count back from the end of the sequences. E.g., my_list[-1] will return the last item in my_list, my_list[-2] will return the second to last, and so on.

Notation	Returns	Examples - if x = [0,1,2,3] Expression will return
x[0]	First element of a sequence	0
x[1]	Second element of a sequence	1
x[-1]	Last element of a sequence	3
x[1:]	Second element through last element	[1,2,3]
x[:-1]	First element up to (but NOT including last element	[0,1,2]
x[:]	All elements - returns a copy of list	[0,1,2,3]
x[0::2]	Start at first element, then every 2nd element	[0,2]

SEQUENCE UNPACKING

Python allows you to return multiple elements of a collection at once:

```
a, b, c = [1, 2, 3]
```

Similar syntax works if we need to assign multiple variables at the same time:

```
a, b, c = lst[1], lst[4], lst[10]
```

Python 3.5 adds a new twist to this syntax and allows you to unpack a part of an original list into a new list:

```
a, *b, c = [1, 2, 3, 4, 5]
# a == 1
# b == [2, 3, 4]
# c == 5
```

Sequence unpacking also works in the `for` loop to iterate over a list of tuples:

```
for a, b in [(1, 2), (3, 4), (5, 6)]:
    print a, b
```

This is especially useful when we need to iterate over keys and values of a dictionary:

```
d = {'key1': 1, 'key2': 2}
for key, value in d.items():
    print(key, value)
```

STRINGS

There are two ways to define strings in Python: either by using single quote characters or by using double quotes characters:

```
s1 = "Hello"
s2 = 'Hello'
```

Both ways are semantically identical and the difference is a matter of a personal preference.

Strings can be used like lists. You can access random characters by index, get the length of a string using the `len` function, and use slicing:

```
s = "hello"
s[1] # 'e'
len(s) # 5
s[1:3] # 'el'
```

In contrast to other programming languages, Python does allow you to concatenate strings with non-string types:

```
"The ultimate answer is: " + 42 # Will cause a runtime error
```

Instead we need to explicitly convert an object into a string first using the `str` function:

```
"The ultimate answer is: " + str(42)
```

Python also allows you to define a multiline string. To do this, we need to start and end a string literal with either three single quote characters or three double quotes characters:

```
m1 = """
Multi
line
"""
m2 = '''
Multi
line
'''
```

Starting from Python 2.6 the idiomatic way to format a string is to use the `format` function:

```
'Good morning, {}. How are you today?'.format('John')
```

This replaced an outdated format operator: `%`.

Python 3.6 has introduced a new syntax for string formatting. Now you can refer variables in the scope if you prepend a string literal with the `f` character:

```
name = 'John'
f'Good morning, {name}. How are you today?'
```

FUNCTIONS

FUNCTION DEFINITIONS

Functions are defined with the `def` keyword and parentheses after the function name:

```
def a_function():
    """ document function here """
    print "something"
```

PARAMETERS

Parameters can be passed in several ways. If we have the following function:

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

The most common way to pass arguments is by position:

```
foo(1, 2, 3)
```

You can also pass arguments by name:

```
foo(b=2, c=3, a=1)
```

Or you can combine both methods. In this case, you need to pass the positional arguments first:

```
foo(1, c=3, b=2)
```

You can also define a function with a variable number of positional parameters:

```
def foo(*args):
    # args is a tuple
    print args
foo(1, 2, 3, 4) # prints (1, 2, 3, 4)
```

Or a variable number of keyword parameters:

```
def foo(**args):
    # args is a dictionary:
    print args
foo(a=1, b=2, c= 3) # prints {'a': 1, 'c': 3, 'b': 2}
```

If you have a list or a dictionary and you want to pass to a function that accepts variable number of parameters, you can use the unpacking feature:

```
def foo(*args, **kw):
    print args, kw
lst = [1, 2, 3]
d = {'a': 1, 'b':2}
foo(*lst, **kw) # prints (1, 2, 3) {'a': 1, 'b': 2}
```

In this case, elements of collections are “unpacked” and passed as separate arguments.

Python 3.6 has introduced syntax for variable annotations that allow you to associate a parameter with a type or any value:

```
def foo(a: int, b: str, c: "param", d: 42):
    print(a, b, c, d)
```

These annotations do not change the semantics of a function, and are intended to be interpreted by third-party libraries.

RETURNING VALUES

You can return any Python object from a function—ints, floats, lists, dictionaries, anything.

```
def foo():
    return dict("color": "blue")
```

Thanks to tuple packing and unpacking, you can also return more than one item at a time. Items separated by commas are automatically 'packed' into a tuple and can be 'unpacked' on the receiving end:

```
def foo():
    return 1,2

a, b = foo()
```

CLASSES

You define a class with the class keyword:

```
class MyClass(object):
    def __init__(self, par):
        # initialize some stuff
        self.foo = "bar"
    def a_method(self):
        # do something
    def another_method(self, parameter):
        # do something with parameter
```

Note: In Python 3.x, you can create classes without inheriting from `object` because that's the default.

Instead of defining getters and setters methods, Python supports defining properties that are defined by methods to get and set a value:

```
class Foo:
    def __init__(self):
        self._bar = 1
    @property
    def bar(self):
        return self._bar
    @bar.setter
    def bar(self, value):
        self._bar = value
foo = Foo()
foo.bar = 5
print foo.bar # 5
```

All methods and fields are “public” in Python, but it is common to start members’ names from “_” if they are not supposed to be used as part of the “public” interface.

INSTANTIATING CLASSES

Classes are instantiated using the class name:

```
my_class_object = my_class()
```

When a class object is instantiated, the class's `__init__(self)` method is called on the instance, usually doing any set up that is needed, such as initializing variables and the like. If the class `__init__( )` method accepts a parameter, it can be passed in:

```
my_class_object = my_class(param)
```

INHERITANCE AND MIXINS

Python supports multiple inheritance. This does provide you with more ways to shoot yourself in the foot, but a common pattern for multiple inheritance is to use "mixin" classes.

ABSTRACT BASE CLASSES, METACLASSES

Abstract base classes are defined in PEP 3119. You can create abstract base classes via the abc module, which was added in Python 2.6.

A metaclass is a class for creating classes. You can see examples of this in Python built-ins, such as int, str, or type. All of these are metaclasses. You can create a class using a specific metaclass via `__metaclass__`. If that is not specified, then type will be used.

COMPREHENSIONS

Python comes with a concept known as comprehension. There are 3 types: list comprehension, dictionary comprehension and set comprehension.

Following is an example of a list comprehension:

```
new_list = [x for x in range(5)]
```

This will create a list from 0-5. It is the equivalent of the following for loop:

```
new_list = []
for x in range(5):
   new_list.append(x)
```

A dictionary comprehension is similar. It looks like this:

```
new_dict = {key: str(key) for key in range(5)}
# new_dict == {0: '0', 1: '1', 2: '2', 3: '3', 4: '4'}
```

A set comprehension will create a Python set, which means you will end up with an unordered collection with no duplicates. The syntax for a set comprehension is as follows:

```
new_set = {x for x in 'mississippi'}
```

You can also filter elements in a list using comprehension:

```
new_list = [x for x in range(10) if x %2 == 0]
# [0, 2, 4, 6, 8]
```

If a comprehension creates a list that is too big or requires significant amount of computation we can use generators:

```
huge_list = (x ** 2 for x in range(1000000))
```

The syntax looks almost the same (parentheses instead of square brackets), but instead of generating all elements in advance it returns an iterator that will generate elements one by one.

STYLE TIPS

WHAT DOES 'PYTHONIC' MEAN?

'Pythonic' is the term that Pythonistas use when they are talking about code that uses the language well and the way that its creators intended. Being Pythonic is a very good thing. Using Java-esque camel-cased variable names is not Pythonic, but using it for class names is. Writing for loops in the style of C/C++ is considered un-Pythonic. On the other hand, using Python data structures intelligently and following the Python style guide makes your code Pythonic.

THE ZEN OF PYTHON

PEP (Python Enhancement Proposal)-20 is the Zen of Python. Written by

long time Python developer Tim Peters, the Zen is acknowledged as the core philosophy of Python. In fact, it is always accessible in any Python environment by using import this:

THE ZEN OF PYTHON, BY TIM PETERS

> *Beautiful is better than ugly.*
> *Explicit is better than implicit.*
> *Simple is better than complex.*
> *Complex is better than complicated.*
> *Flat is better than nested.*
> *Sparse is better than dense.*
> *Readability counts.*
> *Special cases aren't special enough to break the rules.*
> *Although practicality beats purity.*
> *Errors should never pass silently.*
> *Unless explicitly silenced.*
> *In the face of ambiguity, refuse the temptation to guess.*
> *There should be one—and preferably only one—obvious way to do it.*
> *Although that way may not be obvious at first unless you're Dutch.*
> *Now is better than never.*
> *Although never is often better than *right* now.*
> *If the implementation is hard to explain, it's a bad idea.*
> *If the implementation is easy to explain, it may be a good idea.*
> *Namespaces are one honking great idea — let's do more of those!*

PEP-8—THE PYTHON STYLE GUIDE

Python has its own style guide known as PEP8 that outlines various guidelines that are good to follow. In fact, you must follow them if you plan to contribute to Python Core. PEP 8 specifies such things as indentation amount, maximum line length, docstrings, whitespace, naming conventions, etc.

USING THE SHELL

PYTHON'S DEFAULT SHELL

Python is one of several languages that has an interactive shell which is a read-eval-print-loop (REPL). The shell can be enormously helpful for experimenting with new libraries or unfamiliar features and for accessing documentation.

BATTERIES INCLUDED: USING LIBRARIES

IMPORTING AND USING MODULES AND LIBRARIES

Using external modules and libraries is as simple as using the import keyword at the top of your code.

Import	Explanation
`from lib import x` `from lib import x as y`	Imports single element x from lib, no dot prefix needed x() y()
`import lib`	Imports all of lib, dot prefix needed `lib.x()`
`from lib import *`	Imports all of lib, no dot prefix needed. **Not for production code—possible variable name clashes!**

Of the three styles of import, the second (import lib) has the advantage that it is always clear what library an imported element comes from and the chances for namespace collision and pollution are low. If you are only using one or two components of a library, the first style (from lib import x) makes typing the element name a bit easier. The last style (from lib import *) is NOT for production code—namespace collisions are very likely and you can break module reloading. There is one major exception to this rule that you will see in many examples and that concerns the included Tkinter GUI toolkit. Most Tkinter tutorials import it as follow: `from Tkinter import *`. The reason is that Tkinter has been designed so that it is unlikely to cause namespace collisions.

CREATING MODULES AND LIBRARIES

Creating modules and libraries in Python is straightforward. Let's say we have the following file:

```
# utils.py
def foo(a, b):
    return a + b
```

To allow Python to import it we need to put into a directory with the following structure:

```
├── my_library
│   ├── __init__.py
│   ├── utils.py
```

If `my_library` is in the current working directory of an interpreter we can import `utils` in the following way:

```
from my_library import utils
utils.foo(4, 2)
```

Notice that to turn `my_library` into a Python library we need to put an empty `__init__.py` file in it. Otherwise Python won't import modules from a directory.

Be aware that when a module is imported, its code is executed. Therefore, if you want a module to contain an entry point to your application, it is recommended to add the following `if` statement to it:

```
def foo(a, b):
    return a + b
if __name__ == "__main__":
    # Application logic goes here
```

With this you can import functions from a module and safely execute it with a Python interpreter.

THE PYTHON STANDARD LIBRARY—SELECTED LIBRARY GROUPS

Python comes with a standard library of modules that can do much of what you need to get done. The standard library is quite extensive—it would take weeks to become familiar with everything in it.

Whenever you feel the need to go looking for an additional external library, you should first look carefully in the standard library—more often than not, a perfectly good implementation of what you need is already there.

Library Group	Contains Libraries for
File and Directory Access	File paths, tempfiles, file comparisons (see the os and tempfile modules)
Numeric and Math	Math, decimal, fractions, random numbers/sequences, iterators (see math, decimal, and collections)

Library Group	Contains Libraries for
Data Types	Math, decimal, fractions, random numbers/sequences, iterators (see math, decimal, and collections)
Data Persistence	Object serialization (pickle), sqlite, database access
File Formats	CSV files, config files - see ConfigParser
Generic OS Services	Operating system functions, time, command line arguments, logging (see os, logging, time, argparse)
Interprocess	Communication with other processes, low-level sockets (see subprocess and the socket module)
Interned Data Handling	Handling Internet data, including json, email and mailboxes, mime encoding (see json, email, smtplib and mimetools)
Structured Markup	Parsing HTML and XML (see xml.minidom and ElementTree)
Internet Protocols	HTTP, FTP, CGI, URL parsing, SMTP, POP, IMAP, Telnet, simple servers (see httplib, urllib, smtplib, imaplib)
Development	Documentation, test, Python 2 to Python 3 conversion (see doctest and 2to3)
Debugging	Debugging, profiling (see pdb and profile)
Runtime	System parameters and settings, builtins, warnings, contexts (see the dir command and the inspect module)
GUI	Tkinter GUI libraries, turtle graphics

GETTING OTHER LIBRARIES

If you find yourself needing additional functionality, you should go take a look in the Python Package Index (PyPI). There you will find thousands of packages that cover a vast array of topics.

To install the packages, you can use pip or easy_install, both of which you'll need to download from PyPI. For full instructions on bootstrapping with these tools, see pip-installer.org/en/latest/installing.html. Sometimes those utilities won't work and you'll have to use the package's included setup.py to do the installation, which normally goes something like this:

```
python setup.py install
```

You will see a lot of information output to your screen when you execute the above. In some cases, the module has C headers and will require a C/C++ compiler installed on your machine to complete installation correctly.

POPULAR PYTHON LIBRARIES

NUMPY AND SCIPY

NumPy and SciPy are extensive mathematical libraries written to make operating on large data collections easier. As Python's presence in scientific communities has grown, so has the popularity of numpy and scipy. Currently there are conferences devoted to them and to scientific computing. For graphing, you might want to try matplotlib.

IPython – the shell and more The default Python shell has some annoying limitations—it's inconvenient to access the host operating system, there is no good way to save and recover sessions, and it's not easy to export the commands of a session to an ordinary script file. This is particularly irksome for scientists and researchers who may want to spend extensive time exploring their data using an interactive shell.

To address these issues IPython answers these and other problems.

```
C:\Windows\system32\cmd.exe - python ipython
In [1]: print "Hi from IPython!"
Hi from IPython!

In [2]: 1 / 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-b710d87c980c> in <module>()
----> 1 1 / 0

ZeroDivisionError: integer division or modulo by zero

In [3]: import math

In [4]: math.
math.acos      math.copysign  math.floor     math.log       math.sinh
math.acosh     math.cos       math.fmod      math.log10     math.sqrt
math.asin      math.cosh      math.frexp     math.log1p     math.tan
math.asinh     math.degrees   math.fsum      math.modf      math.tanh
math.atan      math.e         math.hypot     math.pi        math.trunc
math.atan2     math.exp       math.isinf     math.pow
math.atanh     math.fabs      math.isnan     math.radians
math.ceil      math.factorial math.ldexp     math.sin

In [4]: math.
```

To get IPython, go to ipython.org and download the version best suited to your operating system.

WEB LIBRARIES

One of the main uses for Python these days is for web programming. There are several popular web frameworks as described below, as well as other libraries for dealing with web content.

DJANGO

Arguably the most popular web framework, Django has taken the Python world by storm in the past few years. It has its own ORM, which makes it very easy to interact with databases.

PYRAMID

A Python framework originally based on Pylons, but is now a rebranding of repoze.bfg. Pyramid supports single file applications, decorator-base config, URL generation, etc.

FLASK

Flask is also a micro web framework for Python, but it is based on Werkzeug and Jinja2.

REQUESTS

Requests is an HTTP library that provides a more Pythonic API to HTTP requests. In other words, it makes it easier to download files and work with HTTP requests than the standard library.

BEAUTIFULSOUP

A great HTML parser that allows the developer to nagivate, search and modify a parse tree as well as dissecting and extracting data from a web page.

OTHER LIBRARIES

Python has many other libraries ranging over all kinds of topics. Here is a sampling:

- Twisted - networking
- Natural Language Tool Kit (NLTK) - language processing
- Pygame - games in Python
- SQLAlchemy - database toolkit

RESOURCES

PYTHON DOCUMENTATION

- Python 3 - docs.python.org/3
- Python 2 - docs.python.org/2.7

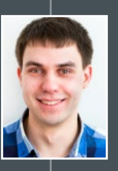
TUTORIALS

- *Official* - docs.python.org/2/tutorial
- *Learn Python* - learnpython.org
- *Learn Python the Hard Way* - learnpythonthehardway.org/book
- *Python Anywhere* - pythonanywhere.com

BOOKS

- *Python 3 Object Oriented Programming* by Dusty Phillips
- *Python Cookbook (2nd Edition) (Python 2.x)* by Alex Martelli, Anna Ravenscroft, David Ascher
- *Python Cookbook (3rd Edition) (Python 3.x)* by David Beazley, Brian K. Jones
- *Python Standard Library* by Doug Hellmann
- *Python in Practice* by Mark Summerfield
- *Dive Into Python* by Mark Pilgrim

ABOUT THE AUTHOR



IVAN MUSHKETYK is a passionate, experienced software development engineer and certified AWS specialist with a vast experience in different domains, including cloud computing, networking, artificial intelligence, and monitoring. Outside of work he is a big Open Source enthusiast and has made several contributions, most notably to Gatling and Apache Flink. He enjoys writing for technical blogs such as SitePoint, DZone, and his blog *Brewing Codes*.

Browse Our Collection of Free Resources, including:

RESEARCH GUIDES: Unbiased insight from leading tech experts

REFCARDZ: Library of 200+ reference cards covering the latest tech topics

COMMUNITIES: Share links, author articles, and engage with other tech experts

JOIN NOW

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

Copyright © 2017 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com