ALSO FROM

DZone

Expert research and learning communities for developers, tech professionals and **smart people everywhere.**

COMMUNITIES

- Share links, write articles, and engage in dialogue with other tech experts
- Topics include Java, Mobile Development, Web Development, SQL, Big Data, Internet of Things, and many more
- Ability to promote your own blog and products through contribution

RESEARCH GUIDES

- Free, unbiased industry insight into key technology topics, trends and vendors
- In-depth articles written by industry experts
- Key findings from our survey of over 1000+ developers and experts
- Profiles and key information on solution providers
- Development checklists and infographic

REFCARDZ

- Access a library of over 200 reference cards covering the latest tech topics, programming languages, and platforms
- Written by industry experts
- Updated monthly

Join DZone today **for free** and gain access to all of these exclusive benefits and more

JOIN NOW

DZone Refcardz

Brought to you by: redhat

CONTENTS INCLUDE:

- Python 2.x vs. 3.x
- Branching, Looping, and Exceptions
- The Zen of Python
- Popular Python Libraries
- Python Operators
- Instantiating Classes... and More!

Core Python

By: Naomi Ceder and Mike Driscoll

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 totem for the language and community, the name actually derives 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 indented it...

Indentation rules in Python. There are no curly braces, no begin and 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 a line with less indentation.

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

```
def function_block():
    # first block
        # second block within first 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 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 3 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 dive in with this example Python script:

```
#! /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.

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, Pythonic style would be more likely to go ahead and try the 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 fairly 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
```

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 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.
`boolean`	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]

FUNCTIONS

Function definitions

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

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

Parameters

Parameters can be passed in several ways:

Default parameters:

```
def foo(a=2, b=3):
    print a
foo()
```

By position:

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

By name:

```
foo(b=4)
```

As a list:

```
def bar(*args):
    print args
bar(1, 2, 3)
```

As a dictionary:

```
def foo(a, b=2, c= 3):
    print a, b, c
d = {a:5, b:6, c:7}
foo(**d)
```

See also keyword arguments (i.e. **kwargs), which allows you to take an arbitrary number of keyword arguments. You can read more about it here: http://docs.python.org/2/tutorial/controlflow.html#keyword-arguments.

Returning values

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

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

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

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

CLASSES

Defining 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. Also don't write getters/setters up front, use the @ property instead which lets you add them transparently later.

Instantiating classes

Classes are instantiated using the class name:

```
my_class_object = my_class()
```

When a class object is instantiated, the classe's __init__(self) method is called on the instance, usually doing any set up that is needed: 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 comprehensions. There are 3 types: list comprehensions, dict comprehensions and set comprehensions.

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 dict comprehension is similar. It looks like this:

```
new_dict = {key: str(key) for key in range(5)}
```

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'}
```

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 it's creators intended. 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 include 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

```
def my_module(foo, bar):
    print foo
    print bar
if __name__ == "__main__":
    my_module(1, 2)
```

Any Python script file can be treated like a module and imported. However be aware that when a module is imported, its code is executed – that's the reason for the if __name__ == "__main__": structure in the example above. In other words, to be safely used as a module, a script should be organized into functions (or classes), with the if statement at the very end.
Here is an example module:

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)
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 http://www.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

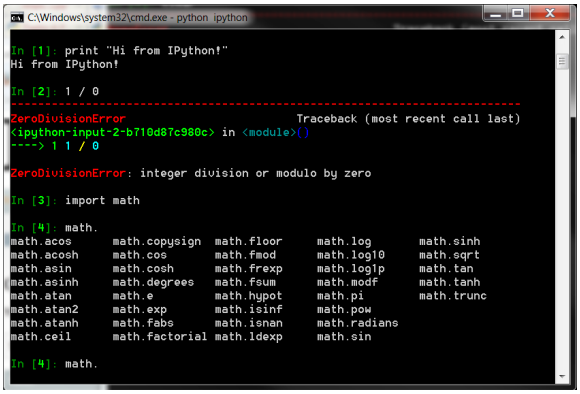
numpy and scipy

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.



To get IPython, go to http://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 - http://docs.python.org/3/

Python 2 - http://docs.python.org/2.7/

Tutorials

Official - http://docs.python.org/2/tutorial/

Learn Python - http://www.learnpython.org/

Learn Python the Hard Way - http://learnpythonthehardway.org/book/

Python Anywhere - https://www.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 Example by Doug Hellmann

Python in Practice by Mark Summerfield

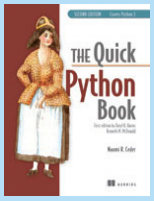
Dive Into Python by Mark Pilgrim

ABOUT THE AUTHOR

Naomi Ceder has been involved in teaching and promoting Python for over a decade. She has taught Python to everyone from 6th graders to adults and is a member of the Python Software Foundation and started both the poster session and the education summit at PyCon. She is also the author of The Quick Python Book, 2nd ed. from Manning Publishers.

Mike Driscoll has been programming in Python since 2006. He enjoys writing about Python on his blog at www.blog.pythonlibrary.org/, occasionally writes for the Python Software Foundation. Mike has also been a technical reviewer for Packt Publishing's Python books, such as Python 3 Object Oriented Programming, and Python 2.6 Graphics Cookbook and more.

RECOMMENDED BOOK



The Quick Python Guide introduces Python's syntax, control flow, and basic data structures, then walks through creating, testing, and deploying full applications and larger code libraries. Includes survey of GUI programming, testing, database access, and web frameworks.

BUY NOW!

DZone

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, blogs, feature articles, source code and more. "DZone is a developer's dream.", says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-80-6
ISBN-10: 1-936502-80-1
50795
9 781936 502806

$7.95