Python Essentials Documentation

Release 0.1a

Jason McVetta

Sep 27, 2017

Contents

1 Work In Progress **1**

2 About **3**
2.1 Introduction 3
2.2 Background Assumptions 3
2.3 Resources 3
2.4 Related Courses 3
2.5 Copying 3

3 Getting Help **5**

4 Data Types **7**
4.1 Numbers 7
4.2 Booleans 10
4.3 None Type 10
4.4 Collections 11
4.5 Strings 16
4.6 Lab - String formatting 20
4.7 Time 21
4.8 Files 21

5 Syntax **23**
5.1 Comments 23
5.2 Operators 23
5.3 Packages 24
5.4 Functions 25
5.5 Scope 27
5.6 Loops 28
5.7 If / Else 29
5.8 Iterators 29
5.9 Generators 30

6 Lab - Echo **33**

7 Classes **35**
7.1 Getters & Setters 36

8 Lab - Dice **39**

8.1 Pseudorandom Number Generator 39

8.2 File Layout 39

8.3 Code Stubs 39

9 Lab - Word Count **43**

CHAPTER 1

Work In Progress

This curriculum is a work in progress. Many sections are missing or incomplete. There may still be some TODOs:

Todo

Rewrite this section with better explanations, and using detailed examples.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-essentials/checkouts/latest/classes.rst, line 6.)

Todo

Rewrite text for this section for better clarity.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-essentials/checkouts/latest/classes.rst, line 108.)

Todo

Add another character to the set math examples, to better illustrate diff between - and ^ operators.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-essentials/checkouts/latest/datatypes/collections.rst, line 352.)

CHAPTER 2

About

Introduction

Welcome to the class notes to *Python Essentials*.

Classroom delivery of this course is available from Silicon Bay Training, who sponsored its development.

Background Assumptions

- Basic familiarity with programming concepts such as variables, loops, functions, etc.

Resources

The source code for these notes can be found on Github.

The latest version of these notes is published at Read the Docs. It is also available in PDF and ePub formats.

Related Courses

Students who have completed *Python Essentials* may wish to check out Advanced Python.

Copying

This curriculum was derived from class notes written by Jeremy Osbourne at Silicon Bay Training.

CHAPTER 3

Getting Help

The interpreter in python can be your best friend, both for experimentation and documentation about Python objects. All of the datatype examples should be run from a python interpreter.

How to get the python version without entering the interpreter:

```
$ python --version
```

How to enter the interpreter from a terminal/console window:

```
$ python
```

How to get help about an object or type (example, the int type). Hit ‘q’ to get out of the help in most python interpreters.

```
help(int)
```

Get a list of all module wide attributes, variables, methods, etc.

```
dir()
```

Note that the output from the dir statement is in the python pretty printed format. The pretty print format will always show a list (square brackets) and all attribute names will be string elements in the list.

Get a list of all attributes on an object or type (example, the int type)

```
dir(int)
```

CHAPTER 4

Data Types

Numbers

Integers

Numbers are immutable.

int

`int` is fixed precision, at least 32 bits.

```
>>> width = 20
>>> height = 5*9
>>> width * height
900

>>> type(900)
>>> type(42) == type(900)
True

>>> int("42")
>>> int('42')
>>> int("""42""")
>>> int("42abc")
>>> int("abc")
>>> int("100", 2)
```

Long

- Unlimited precision[1]

[1] http://docs.python.org/library/stdtypes.html#numeric-types-int-float-long-complex

- Denoted by 'l' or 'L', no difference in case
- **Note:** This is disappearing in Python 3.x. ''int" and longs operate the same in Python 3.x. In other words the max int is essentially boundless.

```
>>> 42l == 42
True
>>> type(42L)
>>> type(42L) == type(42)
False
```

Floating Point Numbers

Float, system dependent precision:

```
>>> 3 * 3.75 / 1.5
7.5

>>> type(4.2)
>>> type(42) == type(42.0)
False

>>> float("42")
>>> float(42)
>>> int(42.0)
```

Float allows us to get access to a special infinity value.

```
>>> i = float("inf")
>>> type(i)
>>> 1000000000000 < i
True
```

Most types and objects, even primitive ones, have object methods and properties.

Need to wrap floating points for this to work.

```
>>> (3.2).is_integer()
False
>>> (3.0).is_integer()
True
```

Complex Numbers

Complex, an example of a type that has a unique creation syntax and object oriented property access. Most people probably won't use complex, but it's a good intro to the subtleties of Python types and built in language mechanics.

```
>>> a = 1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> b = 1.4+0.3j
>>> a + b
(2.9+0.8j)
```

Math

Modulus:

```
>>> 8.0 / 3.0
2.6666666666666665
>>> 8.0 % 3.0
2.0
>>> 8 % 3
2
>>> 9.0 / 3.0
3.0
>>> 9 % 3
0
```

Basic math functions:

```
>>> round(42.01)
42
>>> round(42.01, 2)
42.009999999999998
>>> abs(-42)
42
>>> divmod(42, 2)
(21, 0)
>>> pow(2, 8)
256
```

Many utility functions are available from the `math` library:

```
>>> import math
>>> math.trunc(42.0)
42
>>> math.floor(42.9999)
42.0
>>> math.ceil(42.0001)
43.0
>>> math.trunc(math.ceil(42.0001))
43
>>> math.pi
3.1415926535897931
>>> math.degrees(2*math.pi)
360.0
```

Decimal Class

Not a built in type, but this module is useful for people who need reliable precision with the floating points they use.

We need to import the decimal module:

```
import decimal
```

We'll import the Decimal class by itself for easier use.

```
from decimal import Decimal
```

Now we can use the Decimal type, which defaults to a precision level of 28 digits.

```
>>> Decimal("1") / Decimal("7")
Decimal('0.1428571428571428571428571429')
```

Helps with traditionally tricky, and unreliable, floating point arithmetic.

Note: Here we pass in strings, not floating point numbers. If we pass floating point numbers in, we'll get exact addition from our inexact binary floating point numbers: garbage in, garbage out.

```
>>> 10 + 0.000000000000000001
10.0
>>> Decimal("10") + Decimal("0.000000000000000001")
Decimal('10.000000000000000001')
```

Get the current precision of a decimal, and other settings for the module:

```
>>> decimal.getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,␣
↪capitals=1, flags=[Inexact, Rounded], traps=[InvalidOperation, Overflow,␣
↪DivisionByZero])
```

Booleans

Python has a boolean type, expressed with reserved words `True` and `False`.

```
>>> type(true)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined

>>> type(True)
<type 'bool'>

>>> type(True) == type(1)
False
```

None Type

`None` is Python's null type.

None behaves like one would expact, and it can reliably be tested for, even against things like False and 0.

```
>>> type(None)
<type 'NoneType'>

>>> None == ""
False

>>> None == 0
False

>>> None == False
False
```

```
>>> None == None
True
```

Collections

Lists

Lists are mutable, dynamic arrays.

They are arrays in the sense that you can index items in a list (for example `mylist[3]`) and you can select sub-ranges (for example `mylist[2:4]`). They are dynamic in the sense that you can add and remove items after the list is created.

# To create a list use:

```
>>> items = [111, 222, 333]
>>> items
[111, 222, 333]
>>> len(items)
3
>>> items[0]
111
>>> items[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range


>>> list(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list() takes at most 1 argument (3 given)
>>> list("123")
['1', '2', '3']
>>> list([1, 2, 3])
[1, 2, 3]
>>> list((1, 2, 3))
[1, 2, 3]
```

To add an item to the end of a list, use:

```
>>> items.append(444)
>>> items
[111, 222, 333, 444]
```

To insert an item into a list, use:

```
>>> items.insert(0, -1)
>>> items
[-1, 111, 222, 333, 444]
```

Pop from the right

```
>>> items.pop()
444
```

```
>>> items
[-1, 111, 222, 333]
```

Iterate over items in the list

```
>>> for item in items:
...     print 'item:', item
...
item: -1
item: 111
item: 222
item: 333
```

Other list operations

```
>>> items.count(111)
1
>>> items.index(-1)
0
>>> items.remove(-1)
>>> items
[111, 222, 333]
>>> items.append(111)
>>> items.remove(111)
>>> items
[222, 333, 111]
>>> items.reverse()
>>> items
[111, 333, 222]
>>> items.sort()
>>> items
[111, 222, 333]
```

Check if an item is in the list

```
>>> 111 in items
True
```

Remove an item at a particular index

```
>>> del items[1]
[111, 333]
```

Concatenation comparison

```
>>> ['a'] + ['b']
['a', 'b']
>>> 'a' + 'b'
'ab'
```

Tuples

Tuples are essentially immutable lists.

Helpful when you want to preserve and protect the data, but have the output flexibility of a list.

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345

>>> t
(12345, 54321, 'hello!')
```

Tuples may be nested:

```
>>> u = t, (1, 2, 3, 4, 5)

>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Tuple packing

```
>>> t = 12345, 54321,  'hello!'
>>> t
(12345, 54321, 'hello!')
>>> x, y, z = t
>>> x
12345
>>> y
54321
>>> z
'hello!'
```

Be careful, it is easy to confuse a logical expression and a single-item tuple.

```
>>> astring = ('hello')
>>> astring
'hello'
# Length of the string, not a tuple
>>> len(astring)
5
```

Note the trailing comma

```
>>> a = ('hello',)
('hello',)
>>> len(a)
1
# convert list to tuple
>>> a = tuple(['hello'])
>>> a
('hello',)
```

Sequence packing

```
>>> atuple = 'hello', 'world'
>>> atuple
# Sequencce unpacking
>>> h, w = atuple
>>> h
'hello'
>>> w
'world'
```

Dictionaries

Dictionaries are associative arrays.

Keys are hashable types, like strings and numbers. Tuples can also be keys, as long as they contain only strings, numbers, and more tuples.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}

>>> tel['jack']
4098
>>> len(tel)
2
>>> type(tel)
<type 'dict'>

>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}

>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.values()
[4098, 4127, 4127]
>>> tel.items()
[('jack', 4098), ('irv', 4127), ('guido', 4127)]
>>> type(tel.items())
<type 'list'>
>>> type(tel.items()[0])
<type 'tuple'>

>>> 'guido' in tel
True
>>> 42 not in tel
True
```

Shallow copy

```
>>> hello = tel.copy()
>>> hello
{'guido': 4127, 'irv': 4127, 'jack': 4098}
```

Conversion to a dictionary

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Retrieving key and value in dictionaries

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
..................
gallahad the pure
robin the brave
```

Sets

A set is an unordered collection with no duplicate elements

```
>>> basket = ['apple', 'orange', 'apple', 'pear']
>>> fruit = set(basket)
>>> fruit
set(['orange', 'pear', 'apple'])

>>> 'orange' in fruit
True
>>> 'crabgrass' in fruit
False
```

Set Operators

```
>>> a = set('xyzzyqtt')
>>> b = set('zzytrr')
>>> a
set(['y', 'x', 'z', 'q', 't'])
>>> b
set(['y', 'r', 'z', 't'])
>>> # letters in a but not in b

>>> a - b
set(['q', 'x'])
>>> # letters in either a or b

>>> a | b
set(['q', 'r', 't', 'y', 'x', 'z'])
>>> # letters in both a and b

>>> a & b
set(['y', 'z', 't'])
>>> # letters in a or b but not both

>>> a ^ b
set(['q', 'x', 'r'])
>>> # And a failed operation

>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Todo

Add another character to the set math examples, to better illustrate diff between - and ^ operators.

Strings

Strings are immutable, iterable "lists" of characters in Python. As strings are incredibly important, we will pay a little more attention to them before moving on to the remaining datatypes.

Basics

In python, there is no difference between a single and a double quoted string...

```
>>> s1 = 'abc'
>>> s2 = "abc"
>>> s1 == s2
True
```

Triple quoted, using single or double quotes, is a multiline string.

```
>>> s = '''this is
... a
... multiline
... string
... '''
>>> s
'this is\na \nmultiline\nstring\n'
>>> print s
this is
a
multiline
string

>>>
```

Raw Strings

Raw strings are very useful in Regular Expressions.

```
s4 = r"hello \n world"         # <-- Raw string
```

Unicode Strings

In Python 2.x, strings are ASCII. Starting in Python 3.x, all strings are unicode by default.

```
>>> s = u'Hello\u0020World !'
>>> s
u'Hello World !'
```

`string` module

There is also a string module. Most functions in the module are accessible as instance methods on string types, although there are a few helper attributes that can't be found elsewhere.

```
import string
print string.ascii_letters
#'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
print string.digits
#'0123456789'
print string.punctuation
#'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Operations

Strings can be concatenated with the + operator:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
```

Strings can be repeated with the * operator:

```
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Loops

Accessing a string as an iterable in a for loop:

```
>>> for ch in s1:
...   print ch
..............
a
b
c
d
```

Index Notation

```
>>> word = 'HelpA'

>>> word[4]
'A'

>>> word[0:2]
'He'

>>> word[2:4]
'lp'

>>> word[:2]    # The first two characters
'He'

>>> word[2:]    # Everything except the first two characters
'lpA'

>>> word[-1]     # The last character
```

```
'A'

>>> word[-0]      # (since -0 equals 0)
'H'
```

No reassigning chars:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

Formatting

Justification:

```
>>> "42".rjust(10)
'        42'
>>> "42".center(10)
'    42    '
>>> "42".ljust(10)
'42        '
```

Zero fill padding:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
```

Removing extraneous white space:

```
>>> '   Get Rid of Whitespace, including newlines   \n'.strip()
'Get Rid of Whitespace, including newlines'
>>> '   Get Rid of Whitespace, including newlines   \n'.rstrip()
'   Get Rid of Whitespace'
>>> '   Get Rid of Whitespace, including newlines   \n'.lstrip()
'Get Rid of Whitespace, including newlines   \n'
```

Various methods:

```
>>> "hello, world".split(", ")
['hello', 'world']

>>> ", ".join(["hello", "world"])
'hello, world'
# Or the same thing, written statically with the string library imported.
>>> string.join(["hello", "world"], ", ")
'hello, world'

>>> 'The happy cat ran home.'.upper()
'THE HAPPY CAT RAN HOME.'

>>> 'The happy cat ran home.'.find('cat')
10
```

```
>>> 'The happy cat ran home.'.find('kitten')
-1
```

Modulus (%) Operator

Python uses the % (modulus) operator for formatting (modifying) strings. Within the string to format, a % character marks a token. The %s is the conversion type. If we're passing in a string, use "s".

```
>>> state = 'California'
>>> 'It never rains in sunny %s.' % state
'It never rains in sunny California.'
```

With multiple inputs, we wrap in a tuple:

```
>>> "%s %s!" % ("hello", "world")
'hello world!'
```

Formatting a floating point output:

- zero filled
- 6 total characters (including decimal)
- precision of 3 (which includes all values, not just post decimal values)

```
>>> "%06.3g" % 10.5
'0010.5'
```

Referencing a value from a named attribute instead of a tuple. Can use a tuple or a map, not both:

```
>>> pets = {'dog': 'Fido', 'cat': 'Claude'}
>>> 'My dog is named %(dog)s, and my cat %(cat)s.' % pets
'My dog is named Fido, and my cat Claude.'
```

.format() Method

Repeats reference to first argument:

```
>>> "First, thou shalt count to {0} then to {0}".format(10, 10)
'First, thou shalt count to 10 then to 10'
```

References keyword argument food:

```
>>> "I like {food}".format(food='pizza')
'I like pizza'
```

Accessing class attributes:

```
>>> class Elephant(object):
...     weight = 325
...
...
>>> class Elephant(object):
...     weight = 325
...
>>> e = Elephant()
>>> "Weight in tons {0.weight}".format(e)
'Weight in tons 325'
```

First element of keyword argument `players`:

```
>>> "Units destroyed: {players[0]}".format(players=[80])
'Units destroyed: 80'
```

Referencing items in a dict:

```
>>> d = {"dog": "cat"}
>>> "{0[dog]}".format(d)
'cat'
```

Or with a more shorthand notation:

```
>>> "{dog}".format(**d)
'cat'
```

Implicitly references the first positional argument:

```
>>> "Bring me a {}".format("shoe")
'Bring me a shoe'
```

New style formatting:

```
>>> '{0:<30}'.format('left aligned')
'left aligned                  '
>>> '{0:>30}'.format('right aligned')
'                 right aligned'
>>> '{0:^30}'.format('centered')
'           centered           '
# use '*' as a fill char
>>> '{0:*^30}'.format('centered')
'***********centered***********'
```

Lab - String formatting

Suppose we want to print out a chess or checkerboard using ASCII characters.

Solution:

Let's try it out:

```
$ python formatting.py
X X X X
 X X X X
X X X X
 X X X X
X X X X
 X X X X
X X X X
 X X X X
```

We can also visualize the execution of our program with *Python Tutor*.

Time

Times and dates are not built in types in python, but they're important enough to mention here while we discuss other types.

NOTE: Test these on your system, as there can be differences between unixlike and windows, and there can be differences depending on what is considered localtime, time due to daylight savings, etc.

```
import time

# to get "now" in seconds since the epoch
>>> time.time()
1314852408.951319
# to convert "now" into a human readable time
>>> time.ctime(time.time())
'Wed Aug 31 21:47:01 2011'
# to see the effect of your timezone, call ctime with a value of 1
>>> time.ctime(1)
'Wed Dec 31 16:00:01 1969'


# Others
# Are we in daylight savings? (1 for yes, 0 for no)
>>> time.daylight
1
# To figure out our timezone (timezone is number of seconds west from utc)
>>> -1 * (time.timezone / (60*60))
# Pause the execution of an application for x number of seconds
>>> time.sleep(5)
```

Dates

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{0:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'

### Date arithmetic with timedelta
# assuming datetime has been imported as a full module
>>> now = datetime.datetime.today()
# See the date represented as today
>>> now
datetime.datetime(2011, 11, 24, 12, 40, 57, 433000)
# Add thirty days to today
>>> now += datetime.timedelta(days=30)
# View the new date
>>> now
datetime.datetime(2011, 12, 24, 12, 40, 57, 433000)
```

Files

Input and output of strings to a file:

```
>>> outfile = open('tmp.txt', 'w')
# Note: We must output a newline
>>> outfile.write('This is line #1\n')
>>> outfile.write('This is line #2\n')
>>> outfile.write('This is line #3\n')
>>> outfile.close()
```

Reading an entire file:

```
>>> infile = open('tmp.txt', 'r')
>>> content = infile.read()
>>> print content
This is line #1
This is line #2
This is line #3
>>> infile.close()
```

Reading a file one line at a time:

```
>>> infile = open('tmp.txt', 'r')
>>> for line in infile.readlines():
...     print 'Line:', line
...........................
Line: This is line #1
Line: This is line #2
Line: This is line #3
>>> infile.close()
```

`infile.readlines()` returns a list of lines in the file. For large files use the file object itself `infile.xreadlines()`, both of which are iterators for the lines in the file.

`with` keyword

Files should *always* be closed before the program exits. When using the basic `open()` syntax, it is possible for the programmer to forget to include a `close()` call, or for the program to exit with an exception before the file closes. Using the `with` keyword avoids these pitfalls. The file is open only for the block of code indented beneath the `with` line - when that code exits, the file is closed. This works even if the program exits with an exception.

```
>>> with open('/tmp/foobar.txt', 'w') as outfile:
...     outfile.write('foo\n')
...     outfile.write('bar\n')
...
>>> print outfile
<closed file '/tmp/foobar.txt', mode 'w' at 0x7fdb21d8e390>
>>> with open('/tmp/foobar.txt', 'r') as infile:
...     for line in infile:
...         print line
...
foo

bar

>>>
```

CHAPTER 5

Syntax

Comments

Comments are started with the hash character.

```
a = 1 + 2 # Only things after the hash are ignored.
```

There are no multiline comments in Python. IDEs are your friend when it comes to making multi-line comments.

Operators

Some different forms of operators in Python, to go with the usual C operators everyone is likely familiar with.

The and and or operators do not return a boolean value, instead they return a value that will result in a truthy pass or a falsey fail, and are safe to use anywhere a normal logic check would be used:

```
>>> 1 and 2
2
>>> 1 or 2
1
>>> "" or 0
0
```

Normal logic checks:

```
>>> bool("" or 0)
False
>>> 1 == 2
False
>>> 1 != 2
True
>>> 1 is 2
False
```

```
>>> 1 is not 2
True
```

In Python 3.x, integer division that should return a fraction will return a float.

In Python 2.x, it doesn't:

```
>>> 1 / 2
0
# Results rounded toward minus infinity.
>>> -1 / 2
-1
# Floored operation on quotient, more useful in Python 3.x
# when you don't want a float returned.
>>> -1 // 2
-1
# Easy way to get (quotient, remainder)
>>> divmod(1, 2)
(0, 1)
```

Powers, both valid ways of doing things:

```
>>> 2 ** 8
256
>>> pow(2, 8)
256
```

Python does not have a post- or prefix incrementor or decrementor:

```
>>> a = 1
>>> a++
  File "<stdin>", line 1
    a++
      ^
SyntaxError: invalid syntax
# The proper way to increment or decrement in python is
>>> a += 1
>>> a
2
```

Packages

So far we have been working with fairly short, simple programs contained in single files. That style works great for scripts; but most real applications are more complex, and thus better split among several files.

Package Layout

Consider the file layout of a sample program:

```
foobar/
    __init__.py
    foo.py
    bar/
```

```
        __init__.py
        baz.py
```

`__init__.py`

Python considers any folder that is on the `PYTHONPATH` and contains an `__init__.py` file to be a package, and thus importable.

The `__init__.py` file's primary purpose is to be a marker - any folder containing an `__init__.py` file is an importable package. The file likewise defines the top level of said package's namespace.

Almost *every* Python package contains an `__init__.py` file; and the file name tells us *nothing* about the contents of the file, except that it is package marker. Therefore it is considered inappropriate to put real functionality - class definitions, functions, etc - into an `__init__.py` file.

It is commonplace for `__init__.py` to be left completely empty. If it does contain code, it will typically be limited to docstrings, version constants, and import statements. Since the `__init__.py` file is the top level of it's package's namespace, by importing an object into `__init__.py` we promote that into the module's top level namespace.

Consider what an `__init__.py` file from the example program above might look like:

```
'''
Foobar

A program that puts the foo into bar!  (Caution, use at your own risk.  Your
mileage may vary.)
'''

VERSION = '0.1'

from foo import Spam
from bar.baz import Eggs
```

Importing Packages

In this example, two classes will be available in the top level namespace of the package: `Spam`, which is defined in `foobar/foo.py`; and `Eggs` which is defined in `foobar/bar/baz.py`. Both classes were imported above into `__init__.py`. Consequently code outside this package can import those classes like so:

```
from foobar import VERSION
from foobar import Spam
from foobar import Eggs
# Not imported in __init__.py, so we have to use the full path:
from foobar.bar.baz import Ham
```

Note that in Python, unlike in Java, it is commonplace for multiple class definitions to live in one file. Nor is there necessarily any relationship between the name of a file and the name(s) of the object(s) it contains.

Functions

```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
            print a,
            a, b = b, a+b
```

Now call the function we just defined in various ways:

```
fib(2000)
```

Print the funtion object:

```
print fib
```

Reference the function and call:

```
f = fib
f(100)
```

It's usually better to have a function return a value. Rewrite our funtion above so it's like what follows:

```
def fib(n):
    """Return a list of the Fibonacci series up to n."""
    a, b = 0, 1
    results = []
    while a < n:
            results.append(a)
            a, b = b, a+b
    return results
```

Run it:

```
print fib(1000)
```

Calling the function above without any variables will throw an error:

```
fib()
# You'll get a traceback.
```

But every formal parameter to a function can be given a default value. Rewrite the above as follows (just changing the param):

```
def fib(n=100):
    """Return a list of the Fibonacci series up to n, default
    to 100."""
    a, b = 0, 1
    results = []
    while a < n:
            results.append(a)
            a, b = b, a+b
    return results
```

Now there is no more error:

```
print fib()
```

We can also specifically label an argument:

```
print fib(n=1000)
```

Functions can also take a variable number of unlabeled arguments that, if passed in, will be captured in a list:

```
def echo(*words):
     print "You passed in {0} words.".format(len(words))
     for word in words:
             print "A word: ", word
```

Call the function:

```
echo()
echo("hello", "world")
```

Functions can also take a variable number of labeled arguments that, if passed in, will be captured in a dictionary:

```
def echo2(**words):
     print "You passed in {0} key:value pairs.".format(len(words))
     for k, v in words.items():
             print k, v
```

Call the function, making sure to label the arguments:

```
echo2(hello="world", the="universe")
```

It is possible to combine formal arguments with both the unlabeled variable arguments and the labeled variable arguments. When defining a function, formal arguments must come first, followed by unlabeled variable arguments, followed finally by labeled variable arguments.

Scope

Scope chain is functional in Python:

```
# Global
x = 5
# Function local
def h():
    x = 3
    def q():
        x = 9
        # local q(), should print 9
        print x
    q()
    # local to h(), should print 3
    print x

# Run h
h()
# Global x, should print 5
print x
```

This will reference the global since no local x is defined:

```
x = 5
def f():
    print x
```

This will reference, *and* change the global:

```
x = 5
def f():
    global x
    x = 3
    print x

# After running the above function, do the following
print x
```

Loops

for

```
>>> l = [111, 222, 333, 444]
>>> for item in l:
...     print item
...
111
222
333
444
```

while

Fibonacci series:

```
>>> a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Using Range:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Other range uses:

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
```

Enumerate:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
..................
0 tic
1 tac
2 toe
```

Dictionaries:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
..................
gallahad the pure
robin the brave
```

If / Else

The form of an if...else block is slightly different in Python from other C based languages.

NOTE: There is no switch statement in Python.

```
>>> x = 42
>>> if x < 0:
...     print "less"
... elif x == 0:
...     print "equal"
... else:
...     print "greater"
...
greater
```

Iterators

An iterator allows you to loop over a set of data generated one value at a time. Using an iterator is more memory efficient than storing the entire sequence in memory and iterating over that.

Example:

```
# An iterable returns an iterator object that can be processed in a for loop
class Unicoder():
    # The sole purpose of this class is to build an iterator for
    # testing.
    current = None
    start = 50
    stop = 1000
    def __iter__(self):
        self.current = self.start
        return self
    def next(self):
        if self.current <= self.stop:
            ret = unichr(self.current)
            self.current += 1
            return ret
        else:
            # This step is important.
            # We are _required_ to raise StopIteration in an iterator.
            raise StopIteration()

u = Unicoder()
for char in u:
    print char
```

Generators

Functions that yield a result are essentially an iterable:

```
def abc():
    yield "a"
    yield "b"
    yield "c"
```

When we have yield in a function, it implicitly becomes a “generator” which will act like an iterable:

```
>>> abc()
<generator object abc at 0x7fe30be0a8c0>
>>> for item in abc():
...     print item
...
a
b
c

>>> # We can do it again
>>> for item in abc():
...     print item
...
a
b
c
>>> # To see what's happening
>>> l = abc()
>>> l
<generator object abc at 0x7fe30be0a910>
>>> l.next()
'a'
>>> l.next()
'b'
>>> l.next()
'c'
>>> l.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

# Now let’s generate a sequence of unicode characters, equivalent to what we did in the Iterators section:

```python
# NOTE: Many of these may not be visible characters
def make_unicode():
    for num in range(50, 1000):
        yield unichr(num)

for letter in make_unicode():
    print letter
```

CHAPTER 6

Lab - Echo

Let's create a simple toy that accepts input from the user, and tells the user if they typed in an integer.

```python
'''
Lab - Echo

Write a program that:

* Queries the user for input.
* If the input is an integer, tell the user that they gave us
  an integer, and echo the input.
* If the input is not a number, echo the input.
* If the user input is the word "quit", after doing the above,
  exit the program.
* Repeat until the user quits
'''

def main():
    pass

if __name__ == '__main__':
    main()
```

CHAPTER 7

Classes

Todo

Rewrite this section with better explanations, and using detailed examples.

Classes are template structures in Python. They are special objects and types that allow for the creation of unique copies, or instances, of itself. The most basic type of class is a copy of an empty object.

NOTE: This format is called a “new style class”, as it inherits from at least the Base object type:

```python
class Employee(object):
    # We use pass because, since we have no content, we need something
    # to tell python that the indentation is correct.
    pass
```

Inheritence in Python is achieved by passing a comma delimited list of base classes into a new class. Python classes can exhibit polymorphism and multiple inheritence.

Bosses are an employee, even though they’re the boss:

```python
class Boss(Employee):
    # Define a class attribute.
    title = "Like a boss."
    def __init__(self, name):
        """Init is the special constructor function, and it is designed
        to be overridden and used by the classes we make.

        All class methods in python double as instance methods. They are
        required to have n+1 arguments, where the first argument is always
        the implicit reference to the instance, usually called self.

        When called as a class static method, the caller must pass the
        self reference explicitly, not implicitly.
        """
        # Even though this does nothing, We're going to mimic one way
        # to call a super method in Python.
        super(Boss, self).__init__()
        # Set instance attribute.
        self.name = name

if __name__ == "__main__":
    # Create an empty employee instance.
    peon = Employee()
    # By default, classes are dynamic and we can add attributes to them
    # on the fly (although this can also be turned off with a bit of work).
    peon.name = 'John Peon'
    print "Peon Name:", peon.name

    # Create a boss instance.
    boss = Boss("The Boss")
    # Access instance attribute.
    print "Boss Name:", boss.name
    # Instances reference class attributes implicitly.
    print "Boss Title:", boss.title

    # Check inheritence
    print isinstance(boss, Boss)
    print isinstance(peon, Boss)
    print issubclass(Boss, Employee)
```

Getters & Setters

The decorator idea extends to classes. One useful method is when defining public properties that need to be contrived and don't work well when left as public properties:

```
class Employee(object):
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
    @property
    def name(self):
        """A full name, contrived from two other properties."""
        return self.fname + " " + self.lname
    # If we define the property, we can also define a setter and a
    # deleter, in our example they would be called:
    # @x.setter
    # @x.deleter
    # If we don't define them, the property is not settable
    # and is not deletable.


if __name__ == "__main__":
    e = Employee("like", "a boss")
    # There are no private properties in Python
    print e.fname, e.lname
    # But there are contrived properties
    print e.name
    # The following will be an error since the public interface
    # is not callable.
    #print e.name()
    # The following will also be an error since we didn't define
    # a setter.
    #e.name = "hola"
```

Todo

Rewrite text for this section for better clarity.

CHAPTER 8

Lab - Dice

Let's suppose we want to make a set of dice rolling utilities. We will define a generic `Die` class, representing a single die with an arbitrary number of sides. We will also define six-sided `Die6` and twenty-sided `Die20` subclasses.

Pseudorandom Number Generator

We will use the `randint` function from the `random` library to generate pseudo-random numbers for our dice rolls. It takes two integers as arguments, a min and max, and returns a random integer between the two.

Note: Do NOT use Python's standard `random` library for cryptographic purposes - it is deterministic and NOT a cryptographically secure PRNG.

File Layout

We will start with a file layout like this:

```
dice_lab/
   roll.py
   dice/
      __init__.py
      die.py
      dice.py
```

Code Stubs

roll.py

This script contains an example of using the classes. If you can run the example, your dice classes are at least minimally working.

```
'''
Lab - Dice

Make a package of dice rolling utilities.
* Make a simple Die class for the basic rolling of a die
* Make D6 (six-sided) and D20 (twenty-sided) subclasses
* Each Die instance will have a roll() method, returning a random
  number between 1 and the number of sides on that die.
* When one die instance is added to another die instance, roll() is
  called on both instances and the results added together.
* When a die instance is added to an integer, the instance's roll()
  method is called, and the result is added to the integer.
* If a die is added to something other than an integer or another die, the
  operation should fail with an exception.
'''

from dice.dice import Die6
from dice.dice import Die20


def main():
    d6 = Die6()
    d20 = Die20()
    print d6.roll()      # Returns int between 1 and 6
    print d6 + d20       # Returns int between 2 and 26
    print d6 + 15        # Returns int between 16 and 21
    print d6 + "foobar"  # Throws exception

if __name__ == '__main__':
    main()
```

dice/__init__.py

The `__init__.py` file will be mostly empty, containing only a docstring and a `VERSION` variable:

```
'''
dice - Utilities to simulate dice rolling.
'''

VERSION = '0.1'
```

dice/die.py

The main `Die` class is found in `die.py`:

```
from random import randint

class Die(object):
    '''A single die with arbitrarily many sides.'''

    def roll(self):
        '''Returns a random number between 1 and the number of sides on this die'''
        pass
    def __add__(self, other):
        '''Adds the result of rolling this die to an integer, or to the result of␣
↪rolling
        another die.'''
        pass
```

dice/dice.py

Two subclasses, representing six- and twenty-sided dice, are found in `dice.py`.

```
class Die6(Die):
    '''A six-sided die'''
    pass

class Die20(Die):
    '''A twenty-sided die'''
    pass
```

CHAPTER 9

Lab - Word Count

Let's say we want to find out some stats about a book - the total number of words, and the most frequent words. We will use the `re` and `Counter` classes to facilitate our work.

```
'''
Lab - Book Stats

* Download the book text from:  http://tinyurl.com/sbtrain-2-cities.  You can
  download it manually; no need to make the program do that.
* Report the number of words in the book.
* Report the top ten most used words in the book.
'''

import re
from collections import Counter

def main():
    pass

if __name__ == '__main__':
    main()
```