Python Full Meal

- Matt Harrison
- http://panela.blog-city.com/
- matthewharrison at gmail.com

Note that examples are illustrated as if they were done in a terminal. (This file can also be executed using `doctest.testfile`)

Variables

```
>>> a = 5
>>> b = "A string"
```

Variables aren't "typed" so they can contain anything

```
>>> b = 45.4
>>> b = "another string"
```

Numbers

```
>>> 2 + 2
4
```

Integer division - coerce to float if needed

```
>>> 3 / 4
0
>>> 3 / float(4)
0.75
```

No integer overflow! ('L' for long ints)

```
>>> 4 ** 20
1099511627776L
```

Strings

Specify with single, double or triple quotes

```
>>> hello = 'world'
>>> saying = "ain't ain't a word"
>>> paragraph = """Frank said, "That's
not a way to talk to an economist!"
... Joe replied....
... """
```

Formatting

```
>>> "%d" % 20
'20'
```

```
>>> "%.3f %.2f" % (20, 1/3.) # format
as float with 3 decimal places
'20.000 0.33'
```

Lists, tuples and dictionaries

Lists

```
>>> pets = ["dog", "cat", "bird"]
>>> pets.append("lizard")
>>> pets
['dog', 'cat', 'bird', 'lizard']
```

Tuples

tuples are non-mutable

```
>>> tuple_pets = ("dog", "cat",
"bird")
>>> tuple_pets.append("lizard")
Traceback (most recent call last):
  ...
AttributeError: 'tuple' object has no
attribute 'append'
```

Dictionaries

Dictionaries (also known as hashmaps or associated arrays in other languages)

```
>>> person = {"name": "fred", "age":
29}
>>> person["age"]
29
>>> person["money"] = 5.45
>>> del person["age"]
>>> person
{'money': 5.4500000000000002, 'name':
'fred'}
```

Slicing fun

Indexes

Individual indexes can be picked out of sequences

```
>>> favorite_pet = pets[0]
>>> favorite_pet
'dog'
>>> reptile = pets[-1]
>>> reptile
'lizard'
```

Slicing into a list

Slices can also return lists (and use an optional stride)

```
>>> pets[:2]
['dog', 'cat']
>>> pets[1:]
['cat', 'bird', 'lizard']
>>> pets[::2]
['dog', 'bird']
```

String slicing

Strings (and most sequency things) can be sliced

```
>>> veg = "tomatoe"
>>> correct = veg[:-1]
>>> correct
'tomato'
>>> veg[::2]
'tmte'
>>> veg[::-1] # backwards stride!
'eotamot'
```

Functions

```
>>> def add_5(number):
...     return number + 5
...
>>> add_5(2)
7
```

docstrings

```
>>> def add(number=0, default=6):
...     "add default to number"
...     return number + default
...
>>> add(1)
7
>>> add(30, 40)
70
>>> add(default=2)
2
>>> add(default=3, number=9) # note
order of args
12
```

Whitespace

Instead of using *{* or *}* use a *:* and indent consistently (4 spaces is recommended practice)

Conditionals

```
>>> grade = 95
>>> if grade > 90:
...     print "A"
... elif grade > 80:
...     print "B"
... else:
...     print "C"
...
A
```

Looping

while

```
>>> num = 2
>>> while num > 0:
...     print num
...     num = num - 1
2
1
```

for

```
>>> for num in range(2, 0, -1):
...     print num
2
1
```

break out of loop

```
>>> for num in range(100):
...     print num
...     if num == 1:
...         break
0
1
```

continue

Can `continue` to next item in loop iteration

```
>>> for num in range(10):
...     if num % 2 == 0:
...         continue
...     print num
1
3
5
7
9
```

Importing Libraries

```
>>> import math
>>> math.sin(1)
0.8414709848078965
```

Can also rename imports using `as`

```
>>> import math as math_lib
>>> math_lib.sin(1)
0.8414709848078965
```

Can also import certain parts of libraries

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

File Input/Output

(Importing *tempfile* to create temporary files)

```
>>> import tempfile
```

File output

```
>>> filename = tempfile.mktemp()
>>> fout = open(filename, 'w')
>>> fout.write("foo\n")
>>> fout.write("bar\n")
>>> fout.close()
```

File input

```
>>> fin = open(filename)
>>> lines = fin.readlines()
>>> fin.close()
>>> lines
['foo\n', 'bar\n']
```

(Delete temp file)

```
>>> import os
>>> os.remove(filename)
```

Classes

```
>>> class Animal(object):
...     def __init__(self, name):
...         self.name = name
...
...     def talk(self):
...         print "Generic growl"
...
>>> animal = Animal("thing")
>>> animal.talk()
Generic growl
```

__init__ is a constructor.

```
>>> class Cat(Animal):
...     def talk(self):
...         "Speak in cat talk"
...         print "%s say's 'Meow!'" %
(self.name)


>>> cat = Cat("Groucho")
>>> cat.talk()
Groucho say's 'Meow!'
```

Exceptions

Raising Exceptions

By default the `exceptions` module is loaded into the `__builtins__` namespace (see `dir(__builtins__)`).

```
>>> def add_2(value):
...     if not isinstance(value, (int,
float)):
...         raise TypeError("%s should
be an int or float" % str(value))
...     return value + 2


>>> add_2("foo")
Traceback (most recent call last):
...
TypeError: foo should be an int or
float
```

Catching Exceptions

```
>>> try:
...     [1, 2].remove(3)
... except ValueError, e:
...     print "Removing bad number"
... except Exception, e:
...     print "Generic Error" #
example to show exception chaining
...     # since the previous exception
was caught, this does nothing
... else:
...     # no exceptions
...     print "ELSE"
... finally:
...     # always executes (after else
), so you can cleanup
...     print "DONE"
Removing bad number
DONE
```

Functional Programming

lambda

Create simple functions

```
>>> def mul(a, b):
...     return a * b
>>> mul_2 = lambda a, b: a*b
>>> mul_2(4, 5) == mul(4,5)
True
```

map

Apply a function to items of a sequence

```
>>> map(str, range(3))
['0', '1', '2']
```

reduce

Apply a function to pairs of the sequence

```
>>> import operator
>>> reduce(operator.mul, [1,2,3,4])
24 # ((1 * 2) * 3) * 4
```

filter

Return a sequence items for which `function(item)` is `True`

```
>>> filter(lambda x:x >= 0, [0, -1, 3,
4, -2])
[0, 3, 4]
```

Notes about "functional" programming in Python

- `sum` or `for` loop can replace `reduce`
- List comprehensions replace `map` and `filter`

*args and **kw

```
>>> def param_func(a, b='b', *args,
**kw):
...     print [x for x in [a, b, args,
kw]]
>>> param_func(2, 'c', 'd', 'e,')
[2, 'c', ('d', 'e,'), {}]
>>> args = ('f', 'g')
>>> param_func(3, args)
[3, ('f', 'g'), (), {}]
>>> param_func(4, *args) # tricksey!
[4, 'f', ('g',), {}]
>>> param_func(*args) # tricksey!
['f', 'g', (), {}]
>>> param_func(5, 'x', *args)
[5, 'x', ('f', 'g'), {}]
>>> param_func(6, **{'foo':'bar'})
[6, 'b', (), {'foo': 'bar'}]
```

Decorator Template

```
>>> def decorator(func_to_decorate):
...     # update wrapper.__doc__
and .func_name
...     # or @functools.wraps(wrapper)
...     def wrapper(*args, **kw):
...         # do something before
invocation
...         result =
func_to_decorate(*args, **kw)
...         # do something after
...         return result
...     return wrapper
```

Parameterized decorators (need 2 closures)

```
>>> def limit(length):
...     def decorator(function):
...         def wrapper(*args, **kw):
...             result =
function(*args, **kw)
...             result =
result[:length]
...             return result
...         return wrapper
...     return decorator


>>> @limit(5) # notice parens
... def echo(foo): return foo


>>> echo('123456')
'12345'
```

Class instances as decorators

```
>>> class Decorator(object):
...     # in __init__ set up state
...     def __call__(self, function):
...         def wrapper(*args, **kw):
...             # do something before
invocation
...             result =
self.function(*args, **kw)
...             # do something after
...             return result
...         return wrapper


>>> decorator = Decorator()
>>> @decorator
... def nothing(): pass
```

List Comprehension

```
>>> results = [ 2*x for x in seq \
...             if x >= 0 ]
```

Shorthand for accumulation:

```
>>> results = []
>>> for x in seq:
...     if x >= 0:
...         results.append(2*x)Can be
nested
```

Nested List Comprehensions

```
>>> nested = [ (x, y) for x in
xrange(3) \
...             for y in xrange(4) ]
>>> nested
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0
), (1, 1), (1, 2), (1, 3), (2, 0), (2,
1), (2, 2), (2, 3)]
```

Same as:

```
>>> nested = []
>>> for x in xrange(3):
...     for y in xrange(4):
...         nested.append((x,y))
```

Iteration Protocol

```
>>> sequence = [ 'foo', 'bar']
>>> seq_iter = iter(sequence)
>>> seq_iter.next()
'foo'
>>> seq_iter.next()
'bar'
>>> seq_iter.next()
Traceback (most recent call last):
  ...
StopIteration
```

Making instances iterable

```
>>> class Iter(object):
...     def __iter__(self):
...         return self
...     def next(self):
...         # return next item
```

Generators

Functions with `yield` remember state and return to it when iterating over them

```
>>> def gen_range(end):
...     cur = 0
...     while cur < end:
...         yield cur
...         # returns here next time
...         cur += 1
```

```
>>> print [x for x in gen_range(2)]
[0, 1]
```

Making instances generate

```
>>> class Generate(object):
...     def __iter__(self):
...         # returns a generator
...         return self.next()
...     def next(self):
...         # logic
...         yield result
```

Generator expressions

Like list comprehensions. Except results are generated on the fly. Use (and) instead of [and] (or omit if expecting a sequence)

```
>>> [x*x for x in xrange(5)]
[0, 1, 4, 9, 16]


>>> (x*x for x in xrange(5)) #
doctest: +ELLIPSIS,
<generator object <genexpr> at ...>
>>> list(x*x for x in xrange(5))
[0, 1, 4, 9, 16]
```

Scripts

Instead of having globally executed code create *main* function and invoke it like this

```
>>> def main(arguments):
...     # do logic here
...     # return exit code
>>> if __name__ == "__main__":
...     sys.exit(main(sys.argv))
```

Getting Help

`dir` and `help` or `.__doc__` are your friends

```
>>> dir("") # doctest:
+NORMALIZE_WHITESPACE
['__add__', '__class__',
'__contains__', '__delattr__',
'__doc__',
'__eq__', '__ge__',
'__getattribute__', '__getitem__',
'__getnewargs__', '__getslice__',
'__gt__', '__hash__', '__init__',
'__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__',
'__new__', '__reduce__',
'__reduce_ex__', '__repr__',
'__rmod__',
'__rmul__', '__setattr__', '__str__',
'capitalize', 'center', 'count',
'decode', 'encode', 'endswith',
'expandtabs', 'find', 'index',
'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower',
'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip',
'split',
'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate',
'upper', 'zfill']


>>> help("".split)  # shows
"".split.__doc__
Help on built-in function split:
<BLANKLINE>
split(...)
    S.split([sep [,maxsplit]]) -> list
of strings
<BLANKLINE>
    Return a list of the words in the
string S, using sep as the
    delimiter string.  If maxsplit is
given, at most maxsplit
    splits are done. If sep is not
specified or is None, any
    whitespace string is a separator.
<BLANKLINE>
```

Debugging

``gdb``like debugging is available

Inline literal start-string without end-string.

```
>>> import pdb
>>> #pdb.set_trace() #commented out
for doctest to run
```