Beginning Python

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'
```

New style (3.x)

```
>>> "{0}".format(20)
'20'
>>> "{0:.3f} {1:.2f}".format(20, 1/3.)
'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
```

pass

Can pass to do a no-op

```
>>> if True:
...     pass
```

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']
```

With example

with statement will close files automatically when with block exits (Python 2.5+)

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

(Delete temp file)

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

Classes

In Python 3.x do not need to subclass object.

```
>>> 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. self is a reference to the class instance.

```
>>> 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
```

In Python 3.x (and 2.7) syntax changes (adds as)

```
>>> try:
...     [1, 2].remove(3)
... except ValueError as e:
...     print "Removing bad number"
```

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__',

…

'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

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

Treading on Python

Handout ©2012

The book, *Treading on Python* covers this handout and more in detail

@__mharrison__
http://hairysun.com/books/tread/