Introduction to Python

Demitri Muna
OSU

8 June 2015

Introduction to Python

- No experience with Python is necessary, but we're assuming you've written programs before.
- There are some notable, incompatible differences between Python 2.x and 3.x. It's possible to write code that runs on both.
- You can check which version you have with: `% python --version`
- If you're using 2.x, use at least 2.6, preferably 2.7. There won't be any more major updates of Python 2. Be familiar with both.
- It's reasonable to start new projects with Python 3.0; most packages are compatible.
- Language is continually being updated and modified. More libraries are being added, both in the language and by third parties.
- Try out the examples as we go through them.

Hello World

The simplest application:

I left space to explain the code, but...

```
#!/usr/bin/python

print "Hello world."

```

```
#!/usr/bin/python3

print("Hello world.")

```

tells OS what to run the program with

Save file, run as:

% `python hello_world.py`

or, make it an executable:

% `chmod +x hello_world.py`

% `hello_world.py`

Running Python

interactive mode

```
demitri — python3.4 — 80×24
Last login: Fri Jun  5 20:53:38 on ttys009
blue-meanie [~] % python
Python 3.4.3 |Anaconda 2.1.0 (x86_64)| (default, Mar  6 2015, 12:07:41)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world.")
Hello world.
>>>
```

What if the location of Python changes (e.g. on different servers, using different versions - 2/3)?

```
#!/usr/bin/env python

print("Hello world!")
```

Means: use the first "python" program on my $PATH – use the local environment to choose the Python. This is always recommended.

```
% which python
% type -a python
```

`% which python` — Tells you the first "python" on your $PATH

`% type -a python` — Lists all "python" programs on your $PATH

It's the Future, Today

There is code in 2.x that will break in 3.x, but since they were developed in parallel, the changes were known. Some of the new features/syntax from 3.x can be used in 2.x:

Python 2

```
#!/usr/bin/python2

# no parentheses needed
print "Hello world!"

# ...but work ok
print("Hello world!")
```

Python 3

```
>>> print "Hello world."
  File "<stdin>", line 1
    print "Hello world."
                       ^
SyntaxError: Missing parentheses in call to 'print'
```

">>>" indicates the code was run from the interactive prompt

```
#!/usr/bin/python2

from __future__ import print_function

print("Hello world!")

# this will now fail in Python 2
print "Hello world!"
```

enforces 3.x behavior for print in 2.x

It is *strongly* recommended you use this statement in *all* your Python 2.x code! (It is harmless in 3.x.)

Numbers

Assigning variables, familiar syntax.

Numeric types
- integer
- long integer
- octal (base 8)
- decimal
- complex

"long" integers can be any length!

```
#!/usr/bin/python

# numbers
a = 42
b = 12 + 45

# numeric types
c = 3
d = 3L
e = 027
f = 027.
g = 10j
h = complex(3,5)
print(h.real, h.imag)

print(10/3)
```

comment in Python

Don't write numbers with leading zeros -- they become octal!

Append a "j" to a number to make it complex (engineers use "j", physicists use "i" for $\sqrt{-1}$).

Note: the result is different in 2.x and 3.x (see truncating division).

Numbers

Python operators:

exponent

truncating division

modulo

+	-	*	**	/	//	%
<<	>>	&	|	^	~	
<	>	<=	>=	==	!=	<>

bitwise operators

same, but only use !=

`import`

This command makes an external package available for additional functionality. This one is built into Python.

Note the format of `moduleName.value` (or function)

(This keeps the runtime light since you are only loading the functionality that you use.)

```
#!/usr/bin/python

import sys

# largest integer number on this machine
print(sys.maxint)

# smallest integer on this machine
print(-sys.maxint - 1)
```

You will get a different result running on a 32-bit vs a 64-bit machine (something to be aware of when running your code in different places.)

Truncating Division

In most languages, we find: `10/3` ⟶ `3` operands are integers, result is an integer

Python 2.x

```
>>> 10/3
3
>>> 10//3
3
>>> 10./3.
3.3333333333333333
>>> 10/3.
3.3333333333333333

>>> from __future__ import division
>>> 10/3
3.3333333333333333
>>> 10//3
3
```

another "future" import

Python 3.x

```
>>> 10/3
3.3333333333333333
>>> 10//3
3
```

Again, it is strongly recommended that you use this "future" import in all your 2.x code.

Boolean Values

Boolean values (True/False) are native types in Python.

The capitalization is important.

```
success = True
didFail = False

a = true     # invalid syntax
b = FALSE    # also invalid
```

Strings

Strings can be delimited using single quotes, double quotes, or triple quotes. Use whatever is convenient to avoid having to escape quote characters with a "\".

Strings can be joined together with the "+" operator.

Triple quotes are special in that they let you span multiple lines. Can be three single quotes or three double quotes.

```
# this form
time = "It's five o'clock."

# is better than
time = 'It\'s five o\'clock.'

a = "Ray, when someone asks you \
if you're a god, you say, 'Yes!'"

b = "Roads? Where we're going, " +
    "we don't need roads."

c = "line 1" + "\n" + "line 2"

d = '''this is
all a single string
with the linefeeds included.'''

e = "col 1" + "\t" + "col 2"
```

newline

tab

None

None is a special value that indicates null. Use this, for example, to indicate a variable has not yet been set or has no value rather than some number that has to be "interpreted".

```
# don't do this:
# --------------
mass = -1 # -1 means that
          # the mass has not
          # yet been set
if mass == -1: # ...

# do this instead
# ---------------
mass = None

if mass is None: # ...
```

Containers – Tuples and Lists

Tuples

Groups of items
Can mix types
Can't be changed once created (immutable)

```
a = (1,2,3)
b = tuple() # empty tuple
c = ('a', 1, 3.0, None)
```

Lists

Can mix types
Mutable
Lists, as proper OO objects, have built-in methods.

```
a = [5,3,6,True,[210,220,'a'],5]
b = list() # new, empty list

# add items to a list
b.append(86)
b.append(99)

print(len(b)) # number of items in b

a.sort()     # sort elements in place
a.reverse() # reverse elements in place
a.count(5) # number of times "5" appears in list

print(a.sort()) # returns "None"
print(sorted(a)) # does not modify a
print(sorted(a, reverse=True)) # reverse order
```

Slices

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
print(a[3:5]) # ['d', 'e'], 4th up to 5th item (not inclusive)
print(a[-1])  # last item ('f')
print(a[:3])  # first three items: ['a', 'b', 'c']
print(a[2:])  # all items from 3rd to end: ['c', 'd', 'e', 'f']
print(a[:])   # returns whole list as a copy
```

Containers – Dictionaries

Dictionaries

A group of items that are accessed by a value.

Note: Called hashes or associative arrays in Perl, available as std::map in C++.

Lists are accessed by index - the order is important. To access a given item, you have to know where it is or search for it.

A lot of data aren't inherently ordered. Take ages of people in a family. You don't think "Maggie was the third one born, so must be 1." You mentally map the name to the age.

```
a = [100, 365, 1600, 24]

a[0] # first item
a[3] # 4th item

ages = dict()
ages['Lisa'] = 8
ages['Bart'] = 10
ages['Homer'] = 38

len(ages) # no. of items in dictionary

ages.keys()        # all keys as a list
ages.values()      # all values as a list
del ages['Lisa']        # removes item
ages.has_key('Marge') # returns False
ages.clear()       # removes all values

ages = {'Lisa':8, 'Bart':10, 'Homer':38}
```

shorthand method of creating a dictionary

```
ages[key] = value
```

dictionary name

can be almost any type - numbers, strings, objects (but not lists)

can be any type

Dictionaries are not ordered. You can iterate over them, but the items can be returned in any order (and it won't even be the same twice).

(Compare this idea to the everything box...)

Control Structures

for Loops

In C, we delineate blocks of code with braces – whitespace is unimportant (but good style).

```
void my_c_function {
    # function code here
}
```

In Python, the whitespace is the *only* way to delineate blocks (because it's good style).

```
for simpson in ages.keys():
    print simpson + " is " + str(ages[simpson]) + "years old"

a = 12 # this is outside of the loop
```

You can use tabs *or* spaces to create the indentation, but you cannot mix the two. Decide which way you want to do it and stick to it. People debate which to use (and if you can be swayed, I like tabs...).

Example: Given an array *a* of 10 values, print each value on a line.

C/C++

```
# given a list of 10 values
for (int i=0;i<10;i++) {
    value = a[i]
    printf ("%d", value)
}
```

Python

```
for value in a:
    print value
```

Can be anything in the list, and can create them on the fly:

```
for string in ['E', 'A', 'D', 'G', 'B', 'e']:
    # do something
```

Control Structures

If you *do* need an index in the loop:

```
a = ['a', 'b', 'c', 'd', 'e']:
for index, item in enumerate(a):
    print index, item

# Output
# 0 a
# 1 b
# 2 c
# 3 d
# 4 e
```

if statement

```
if expression1:
    # statement 1
    # statement 2
elif expression2:
    pass
elif expression3:
    ...
else:
    statement 3
    statement n
```

expressions are Boolean statements

while loop

```
# How many times is this
# number divisible by 2?
value = 82688
count = 0
while not (value % 2):
    count = count + 1
    value = value / 2
    print value
print count
```

continue skips to the next item in the loop

```
for item in some_list:
    if skip_item(item):
        continue
    # process the item
```

break exits the loop immediately

```
for item in some_list:
    if danger_will_robinson:
        break # exit loop
    print "Proceed, Robot"
```

turning on/off blocks of code can be useful for debugging; set to False when done

```
if True:
    # debug statements
    # print stuff
```

Printing Variables

format method on strings

```
a = 12.2
b = 5
c = [a,b,42]
a_dict = {"tiger":"Hobbes", "boy":"Calvin", "philosopher":"Bacon"}

print("The value of a is: {0}".format(a))
print("The value of a is {0} and the value of b is {1}".format(a,b))
print("First and second elements of array: {0[0]}, {0[1]}".format(c))
print("A {0[boy]} and his {0[tiger]}.".format(a_dict))
print("Formatted to two decimal places: {0:.2f}, {1:.2f}".format(a, b))
print("Pad value to 10 characters: {0:10}".format(a))
print("Cast value to string: {0!s}".format(a)) # same as ...format(str(a))
```

first item in format list

second item

this is a tuple

Older '%' style, shown since you'll come across it, but recommend *format*.

```
a = 12.4 # type is float (f)
b = 5 # type is integer (d = decimal)

print("The value of a is: %f" % a)
print("The value of a is %f and the value of b is %d" % (a,b))

# Format float output:
print("The value of a is: %.3f" % a) # three decimal places
```

Note the need for parentheses with more than one value.

This is standard `printf` style formatting - google "printf format" for examples

Files

Open a file

```
filename = "rc3_catalog.txt"
f = open(filename)
rc3_catalog_file = open(filename)
# read file
rc3_catalog_file.close()
```

bad style - be descriptive in your variable names!

The actual filename is an input to your program. Try to abstract your inputs and place them at the top of the file.

Code defensively – what if the file isn't there? You'll be surprised how much time this will save you.

```
try:
    rc3_catalog_file = open(filename)
except IOError:
    print("Error: file '{0}' could not be opened.".format(filename))
    sys.exit(1)
```

- Minimize how much you put in the `try:` block.
- Determine what the error would be by making the code fail in a simple program.

Files

Read over all of the lines in the file:

```
for line in rc3_catalog_file:
    if line[0] == '#':
        continue
    line = line.rstrip("\n")
    values = line.split()
rc3_catalog_file.close()
```

skip lines that begin with a '#'

strip the newline character from each line (split also removes \n)

separate the values by whitespace and return as an array

Write to another file:

```
output_file = open("output_file", mode="w")
output_file.write(a,b)
output_file.close()
```

"w" makes file writeable - will delete existing file!!

File modes:

`r` : read-only (default)
`w` : write, truncate (i.e. empty) file if exists
`a` : if file exists, append to it, create new otherwise
`b` : treat file as binary (i.e. not text), used for image data, etc.

try/except

```
import sys

a = 1
b = 0

print a / b

# Result:
# ZeroDivisonError: integer division or modulo by zero

try:
    c = a / b
except ZeroDivisionError:
    print "Hey, you can't divide by zero!"
    sys.exit(1) # exit with a value of 0 for no error, 1 for error
```

You don't have to exit from an error – use this construct to recover from errors and continue.

```
try:
    c = a / b
except ZeroDivisionError:
    c = 0

# program continues
```

```
# check if a dictionary has
# a given key defined
try:
    d["host"]
except KeyError:
    d["host"] = "localhost"

# Although, this command does the same thing:
d.get("host", default="localhost")
```

try/except

called only when `try` succeeds

provides the opportunity to clean up anything previously set up – always called

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

(From the Python documentation.)

with

A common pattern:

```
# set things up
try:
    # do something
except SomeError:
    # handle error
else:
    # if no error occurred
finally:
    # clean up regardless of path
```

want to close file whether there was an error or not

Example:

```
datafile =
open("filename.txt")
try:
    data = datafile.read()
except SomeError:
    # handle error
finally:
    datafile.close()
```

```
with open("filename.txt") as datafile:
    data = datafile.read()
```

- The file is automatically closed at the end of the block, even if there was an error.
- The file is only defined in the block.
- This extra functionality is built into the object.
- The `with` statement isn't *that* common, and it's not trivial to write your own. But there are times it's useful.

Casting

Where appropriate, you can convert between types:

```
a = "1234" # this is a string
b = int(a) # convert to an integer

# but to be safer...

try:
    b = int(a)
except ValueError:
    b = None
```

Other examples:

```
a = '12.3e4'

print float(a) # 123000.0

print complex(a) # (123000+0j)

#print int(a) # ValueError

print int(float(a)) # 123000

print bool(a) # True

print str(complex(a)) # (123000+0j)
```

Code Defensively – asserts

As your program runs, you make certain assumptions about your code. For example, we have an array that some process fills, and we assume it won't be empty.

```
my_values = list()
# some code to populate my_values

assert len(my_values) > 1, "my_values was empty!"
for i in my_values:
    # do stuff
```

If my_values is empty, this loop is skipped silently.

If this fails, then the exception AssertionError is thrown and this message is printed out.

Be liberal with `assert` statements - they cost nothing. When your script is ready for production use, you can turn them off in two ways:

header in file

```
#!/usr/bin/env python -O
```

command line

```
% python -O myScript.py
```

Can perform more than one check:

```
assert a > 10 and b < 20, "Values out of range."
```

List Comprehension

Take the numbers 1-10 and create an array that contains the square of those values.

One of the nicest features of Python!

List comprehension generates a new list.

```
a = range(1,10+1)

a2 = list()
for x in a:
    a2.append(x**2)

a2 = [x**2 for x in a]
```

Using a for loop

Using list comprehension

Can also filter at the same time:

```
a = range(1,50+1)
# even numbers only
b = [x for x in a if x % 2 == 0]
```

Convert data types:

```
# read from a file
a = ['234', '345', '42', '73', '71']
a = [int(x) for x in a]
```

Call a function for each item in a list:

```
[myFunction(x) for x in a]
```

can ignore return value (which is a list)

Functions / Methods

document function with triple-quoted string

```
def myFormula(a, b, c, d):
    ''' formula: (2a + b) / (c - d) '''
    return (2*a + b) / (c - d)
```

indent as with loops

can set default values on some, all, or no parameters

```
def myFormula(a=1, b=2, c=3, d=4):
    ''' formula: (2a + b) / (c - d) '''
    return (2*a + b) / (c - d)

print myFormula(b=12, d=4, c=5)
```

Note order doesn't matter when using the names (preferred method).

If a default value is set, you don't have to call it at all.

Useful math tools:

```
import math

# constants
a = math.pi
b = math.e

c = float("+inf")
d = float("-inf")
e = float("inf")
f = float("nan") # not a number

def myFormula(a, b, c, d):
    ''' formula: (2a + b) / (c - d) '''
    num = 2 * a + b
    den = c - d
    try:
        return num/den
    except ZeroDivisionError:
        return float("inf")

# tests
math.isnan(a)
math.isinf(b)
```

Functions / Methods

Passing parameters into function / methods.

Unlike C/C++, the parameter list is dynamic, i.e. you don't have to know what it will be when you write the code.

You can also require that all parameters be specified by keywords (kwargs).

Accepts any number of arguments (of any type!)

```
def myFunction(*args):
    for index, arg in enumerate(args):
        print "This is argument {0}: {1}".format(index+1, str(args[index]))

myFunction('a', None, True)

# Output:
# This is argument 1: a
# This is argument 2: None
# This is argument 3: True
```

Note two '**' here vs. one above.

kwargs = keyword arguments

```
def myFunction2(**kwargs):
    for key in kwargs.keys():
        print "Value for key '{0}': {1}".format(key, kwargs[key])

myFunction2(name="Zaphod", heads=2, arms=3, president=True)

# Output:
# Value for key 'president': True
# Value for key 'heads': 2
# Value for key 'name': Zaphod
# Value for key 'arms': 3
```

Note the output order is not the same (since it's a dictionary).

Can be mixed:

```
def myFunction3(*args, **kwargs):
    print "ok"

myFunction3()
myFunction3(1, 2, name="Zaphod")
myFunction3(name="Zaphod")
myFunction3(name="Zaphod", 1, True)
```

zero args are ok

Invalid - named arguments must follow non-named arguments (as defined).

Odds and Ends

Range

```
range(10)       # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(10,20)    # [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
range(10,20,2) # [10, 12, 14, 16, 18]
```

useful in loops

(start, stop, step) - step can only be an integer

```
[x * 0.1 for x in range(0,10)]
```

generate ranges in non-integer steps

Objects and Copies

Does not make a copy – these are the same objects!

Copies all of the items into a new object.

```
ages = {'Lisa':8, 'Bart':10, 'Homer':38}
simpsons = ages
ages['Bart'] = 9
print simpsons['Bart'] # output: 9

ages = {'Lisa':8, 'Bart':10, 'Homer':38}
simpsons = ages.copy()
ages['Bart'] = 9
print simpsons['Bart'] # output: 10

simpsons = dict(ages) # also makes a copy
```

Odds and Ends

The *in* operator:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
print 'a' in a      # True
print 'x' not in a # True
```

Comparison operators can be chained:

```
if 0.1 < x < 3.1:
    # number is in range
```

Create Strings from Lists with a Delimiter

```
strings = ['E', 'A', 'D', 'G', 'B', 'e']
print "|".join(strings)
# Output: E|A|D|G|B|e
```

Importing Packages

```
import math
print(math.pi)

# 3.14159265359
```

"pi" is defined in the "math" package. Access it by specifying the module, then the value (or function).

"pi" is not defined by calling import alone

```
import math
print(pi)

# Traceback (most recent call last):
#   File "untitled text 54", line 2, in <module>
#     print pi
# NameError: name 'pi' is not defined
```

```
from math import *

d = 89
e = 20
f = 297
# ... lots of code
print(e**2)
# 400, not 2.71828182846^2
```

bring everything in math into our namespace

overwrites the 'e' variable from math

The *namespace* is the context where variables are defined. Your script has a namespace. Each module has an independent namespace.

```
import math

>>> pi = 3 # Indiana pi
>>> print(pi)
# 3

>>> print(math.pi)
# 3.14159265359
```

our namespace

the "math" module namespace

```
from math import pi

print(pi)
# 3.14159265359
```

bring "pi" into our namespace - no "math." prefix needed

"`import *`" is bad form and can easily lead to errors. Don't use it unless you really know what you're doing (it's bad style).

Python 2 vs 3

Python 2.7 is the last major release of Python, released in 2010. That means it's been years since new features have been added to the language. Python 3 is ready for use.

Python 2.7 will be maintained until 2020 (was to be 2015, but extended).

If you use Python 2.7, use these imports in all your code to simplify upgrading in the future:

```
from __future__ import division
from __future__ import print_function
from __future__ import absolute_import
```

More information about absolute imports:

https://docs.python.org/2.5/whatsnew/pep-328.html
http://blog.tankywoo.com/python/2013/10/07/python-relative-and-absolute-import.html

Python's Paths

When you `import` a package (or file), how does Python know where to find it? Python first looks in the same directory as the script being run. Next Python has a path list similar to the Unix shell's `$PATH` environment variable that it checks. You can see what this is with:

```
import sys
print(sys.path)
```

You can add your own paths at runtime like this (since it's just a regular list):

```
import sys
sys.path.append("/home/me/lib/python")
```

New directories can be added in the Unix shell via the `$PYTHONPATH` environment variable:

```
% export PYTHONPATH=$PYTHONPATH:$HOME/lib/python
```

This is useful when you write your own modules. Create a directory and put your custom library into it, then add it to `$PYTHONPATH`. If your code is in version control, add those directories to `$PYTHONPATH`.

Further Reading

This is a great reference for Python 2.7. Keep this bookmark handy.

http://rgruet.free.fr/PQR27/PQR2.7.html

Several people have emailed me this – it's also a good introduction.

http://www.greenteapress.com/thinkpython/thinkCSpy/html/

This web page has over one hundred "hidden" or less commonly known features or tricks. It's worth reviewing this page at some point. Many will be beyond what you need and be CS esoteric, but lots are useful. StackOverflow is also a great web site for specific programming questions.

http://stackoverflow.com/questions/101268/hidden-features-of-python

And, of course, the official Python documentation:

http://docs.python.org

Finally, if you are not familiar with how computers store numbers, this is mandatory reading:

http://docs.python.org/tutorial/floatingpoint.html