Introduction to Python

Prof. James H. Thomas

Use python interpreter for quick syntax tests.

Write your program with a syntax-highlighting text editor.

Save your program in a known location and using ".py" extension.

Use the command window (or terminal session) to run your program (make sure you are in the same directory as your program).

Getting started on the Mac

- Start a terminal session
- Type "python"
- This should start the Python interpreter (often called "IDLE")
- Use the Python interpreter to test simple things.

```
> python
Python 2.6.4 (something something)
details something something
Type "help", "copyright", "credits" or "license"
for more information.
>>> print "Hello, world!"
Hello, world!
```

Run your program

- In your terminal, Ctrl-D out of the python interpreter (or start a new terminal).
- Type "pwd" to find your present working directory.
- Open TextWrangler.
- Create a file with your program text.
- Be sure that you end the line with a carriage return.
- Save the file as "prog.py" in your present working directory.
- In your terminal, type "python prog.py"

```
> python hello.py
hello, world!
```

Common beginner's mistakes

If your terminal prompt has three '>' characters you are in the Python interpreter:

```
>>> print 7
7
>>>
```

To run a program, be sure you have a normal terminal prompt (will vary by system), will usually end with a '$' or a single '>' character:

```
>>> python myprog.py arg1 arg2
(program output)
```

When you write your program (in a text editor), be sure to save it before trying out the new version! Python reads the saved file to run your program.

Summary of Command Line Basics

Run a program by typing at a terminal session command line prompt (which may be `>` or `$` or something else depending on your computer; it also may or may not have some text before the prompt).

If you type '`python`' at the prompt you will enter the Python IDLE interpreter where you can try things out (ctrl-D to exit).

If you type '`python myprog.py`' at the prompt, it will run the program '`myprog.py`' if it is present in the present working directory.

'`python myprog.py arg1 arg2`' (etc) will provide command line arguments to the program.

Each argument is a string object and they are accessed using `sys.argv[0]`, `sys.argv[1]`, etc., where the program file name is the zeroth argument.

Write your program with a text editor and be sure to save it in the present working directory before running it.

Objects and types

- An object refers to any entity in a python program.
- Every object has an associated type, which determines the properties of the object.
- Python defines six types of built-in objects:

Number	10 or 2.71828
String	“hello”
List	[1, 17, 44] or [“pickle”, “apple”, “scallop”]
Tuple	(4, 5) or (“homework”, “exam”)
Dictionary	{“food” : “something you eat”, “lobster” : “an edible arthropod”}
File	more later…

- It is also possible to define your own types, comprised of combinations of the six base types.

Literals and variables

- A variable is simply a name for an object.
- For example, we can assign the name "pi" to the Number object 3.14159, as follows:

```
>>> pi = 3.14159
>>> print pi
3.14159
```

- When we write out the object directly, it is a literal, as opposed to when we refer to it by its variable name.

The command line

- The command line is the text you enter after the word "python" when you run a program.

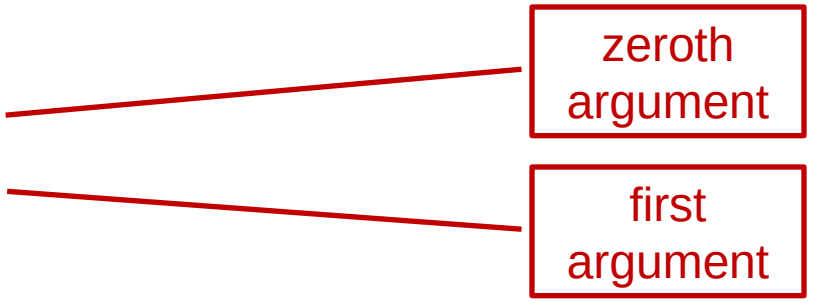
```
python my-program.py GATTCTAC 5
```

- The zeroth argument is the name of the program file.
- Arguments larger than zero are subsequent elements of the command line.

Reading command line arguments

Access in your program like this:

```
import sys
print sys.argv[0]
print sys.argv[1]
```




```
> python my-program.py 17
my-program.py
17
```

There can be any number of arguments, accessed by sequential numbers (sys.argv[2] etc).

Assigning variables

In order to retain program access to a value, you have to assign it to a variable name.

```
import sys
```

this says "give me access to all the stuff in the sys module"

```
sys.argv[0]
```

this doesn't do anything – it says "get the string that is stored at index 0 in the list sys.argv and do nothing with it"

```
import sys
print sys.argv[0]
```

this says "get the string that is stored at index 0 in the list sys.argv and print it" (but it doesn't do anything else)

```
import sys
s = sys.argv[0]
```

this says "get the string that is stored at index 0 in the list sys.argv and assign it to the variable s"

Numbers

- Python defines various types of numbers:
  - Integer (1234)
  - Floating point number (12.34)
  - Octal and hexadecimal number (0177, 0x9gff)
  - Complex number (3.0+4.1j)
- You will likely only need the first two.

Conversions

```
>>> 6/2
3
>>> 3/4
0
>>> 3.0/4.0
0.75
>>> 3/4.0
0.75
>>> 3*4
12
>>> 3*4.0
12.0
```

truncated rather than rounded

- The result of a mathematical operation on two numbers of the same type is a number of that type.
- The result of an operation on two numbers of different types is a number of the more complex type.

integer → float

Formatting numbers

- The % operator formats a number.
- The syntax is `<format> % <number>`

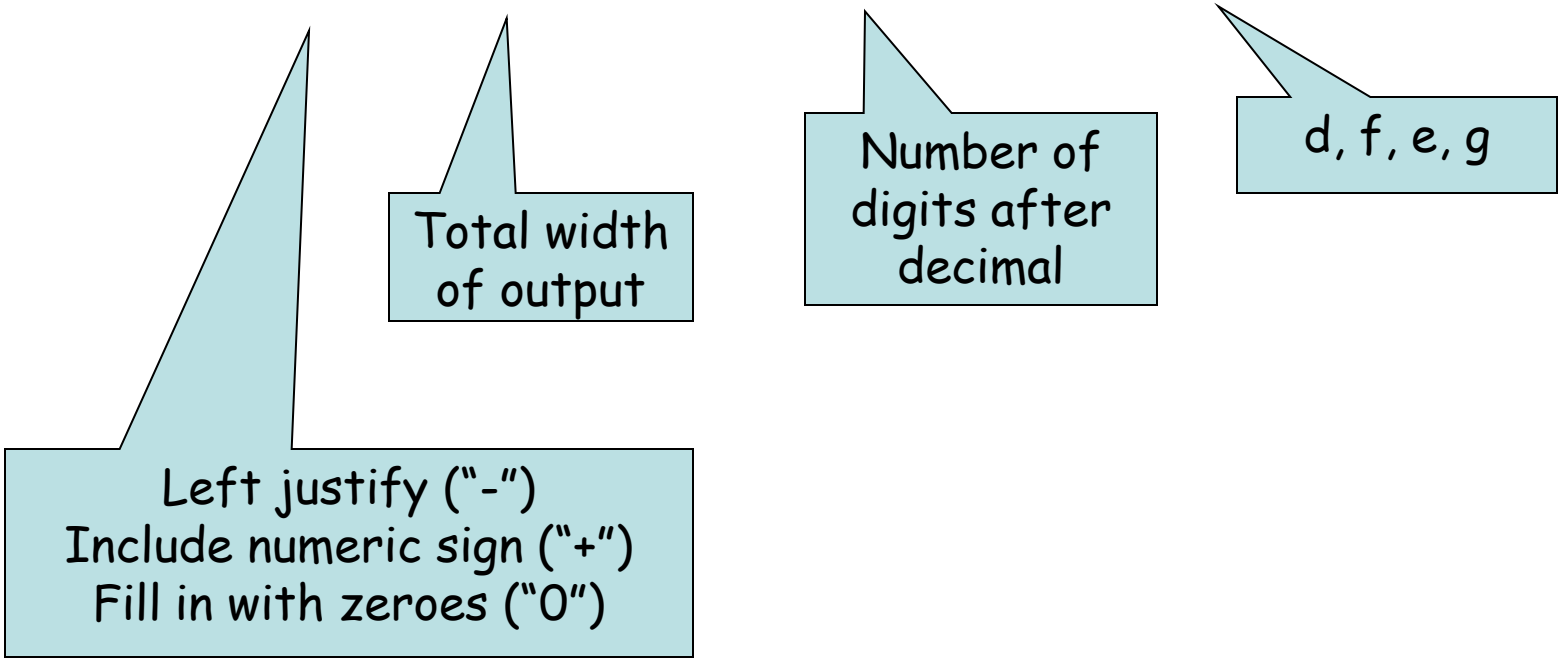
```
>>> "%f" % 3
'3.000000'
>>> "%.2f" % 3
'3.00'
>>> "%5.2f" % 3
' 3.00'
```

Formatting codes

- %d = integer (d as in digit)
- %f = float value - decimal (floating point) number
- %e = scientific notation
- %g = easily readable notation (i.e., use decimal notation unless there are too many zeroes, then switch to scientific notation)

More complex formats

%[flags][width][.precision][code]



Examples

```
>>> x = 7718
>>> "%d" % x
'7718'
>>> "%-6d" % x
'7718  '
>>> "%06d" % x
'007718'
>>> x = 1.23456789
>>> "%d" % x
'1'
>>> "%f" % x
'1.234568'
>>> "%e" % x
'1.234568e+00'
>>> "%g" % x
'1.23457'
>>> "%g" % (x * 10000000)
'1.23457e+07'
```

Read as "use the preceding code to format the following number"

Don't worry if this all looks like Greek – you can figure out how to do these when you need them in your programs.

Ιτ συρε λοοκσ λικε Γρεεκ το με.

string basics

Basic string operations:

```
S = "AATTGG"          # assignment - or use single quotes ' '
S1 + S2               # concatenate two strings
S*3                   # repeat string S 3 times
S[i]                  # get character at position 'i'
S[x:y]                # get a substring from x to y (not including y)
len(S)                # get length of string
int(S)                # turn a string into an integer
float(S)              # turn a string into a floating point decimal number
len(S[x:y])           # the length of s[x:y] is always y - x
```

Methods:

```
S.upper()             # convert S to all upper case, return the new string
S.lower()             # convert S to all lower case, return the new string
S.count(substring)    # return number of times substring appears in S
S.replace(old,new)    # replace all appearances of old with new, return the new string
S.find(substring)     # return index of first appearance of substring in S
S.find(substring, index)      # same as previous but starts search at index in S
S.startswith(substring)       # return True or False
S.endswith(substring)         # return True of False
```

Printing:

```
print var1,var2,var3          # print multiple variables with space between each
print "text",var1,"text"      # print a combination of explicit text and variables
```

list basics

Basic list operations:

```
L = ['dna','rna','protein']     # list assignment
L2 = [1,2,'dogma',L]            # list can hold different object types
L2[2] = 'central'               # change an element (mutable)
L2[0:2] = 'ACGT'                # replace a slice
del L[0:1] = 'nucs'             # delete a slice
L2 + L                          # concatenate
L2*3                            # repeat list
L[x:y]                          # get a slice of a list
len(L)                          # length of list
''.join(L)          # convert a list to a string (a string function that acts on lists)
S.split(x)                      # convert string to list- x delimited
list(S)                         # convert string to list - explode
list(T)                         # converts a tuple to list
```

Methods:

```
L.append(x)                     # add to the end
L.extend(x)                     # append each element from x to list
L.count(x)                      # count the occurrences of x
L.index(x)                      # get element position of first occurrence of x
L.insert(i,x)                   # insert element x at position i
L.remove(x)                     # delete first occurrence of x
L.pop(i)                        # extract (and delete) element at position i
L.reverse()                     # reverse list in place
L.sort()                        # sort list in place
```

`dict` basics

```
D = {'dna':'T','rna':'U'}   # dictionary literal assignment
D = {}                      # make an empty dictionary
D.keys()                    # get the keys as a list
D.values()                  # get the values as a list
D['dna']                    # get a value based on key
D['dna'] = 'T'              # set a key:value pair
del D['dna']                # delete a key:value pair
D.pop('dna')                # remove key:value (and return value)
'dna' in D                  # True if key 'dna' is found in D, else False
```

The keys must be immutable objects (e.g. string, int, tuple).

The values can be anything (including a list or another dictionary).

The order of elements in the list returned by `D.keys()` or `D.values()` is arbitrary (effectively random).

File reading and writing

The `open()` command returns a file object:

```
<file_object> = open(<filename>, <access type>)
```

Access types: `'r'` = read
`'w'` = write
`'a'` = append

`myFile = open("data.txt", "r")` - open for reading
`myFile = open("new_data.txt", "w")` - open for writing
`myString = myFile.read()` - read the entire text as a string
`myStringList = myFile.readlines()` - read all the lines as a list of strings
`myString = myFile.readline()` - read the next line as a string
`myFile.write("foo")` - write a string (does not append a newline)
`myFile.close()` - always close a file after done

if - elif - else

```
if <test1>:
   <block1>
elif <test2>:
   <block2>
elif <test3>:
   <block3>
else:
   <block4>
```

- Only one of the blocks is ever executed.
- A block is all code with the same indentation.

Comparison operators

- Boolean: `and, or, not`
- Numeric: `< , > , ==, !=, >=, <=`
- String: `in, not in`

```
<     is less than
>     is greater than
==    is equal to
!=    is NOT equal to
<=    is less than or equal to
>=    is greater than or equal to
```

`for` loops

`for <target> in <object>` object can be a list, a string, a tuple

`for letter in "Constinople"`

`for myString in myList` (where myList is a list of strings)

`continue` skip the rest of the loop and start at the top again

`break` quit the loop immediately

As usual, all the commands with the same indentation are run as a code block.

`for integer in range(12)` range simply returns a list of integers

`range([start,] stop [,step])`

Loops can be nested or have other complex code blocks inside them.

Examples of `for` loops

```
for base in sequence:
    <do something with each base>

for sequence in database:
    <do something with each sequence>

for base in ["a","c","g","t"]:
    <do something with each base>

for index in range(5,200):
    <do something with each index.
```

`while` loops

Similar to a `for` loop

```
while (conditional test):
      <statement1>
      <statement2>
      . . .
      <last statement>
```

While something is `True` keep running the loop, exit as soon as the test is `False`.

Any expression that evaluates True/False can be used for the conditional test.

Examples of `while` loops

```
while (error > 0.05):
     <do something that will reduce error>
```

```
while (score > 0):
     <traceback through a DP matrix, each
     time setting the current score>
```

Time efficiency

Rough order of speed for common operations:

reading/writing files - very slow
going through a list serially for matching elements - slow
accessing a list (or string) element by index - fast
accessing a dictionary value by key - fast

File reading - sometimes you only need to look through a file until you find something. In this case, read lines until you have what you need then close the file.

Dictionaries can be used in various clever ways to save time.

Do simple profiling to see what part of your code is slowest (for example, invoke `time.time()` twice, once before and once after a code block).

Future - beginning Python is kept simple by hiding a lot of complex things from you - dig in deeper to understand what takes more time (and memory).

Memory efficiency

File reading - often you don't need to save the entire contents of a file into memory. Consider whether you can discard some of the information.

If your program generates many copies of a long string, consider making a dictionary entry with the string as the value (you only need to store it once).

If you are working with a long string and you want to access many segments of it, do NOT save each segment as a string - use string indexing to get each segment as you need it.

Future - instead of using Python lists, consider using classes in the Python array module to store long sequences, etc.