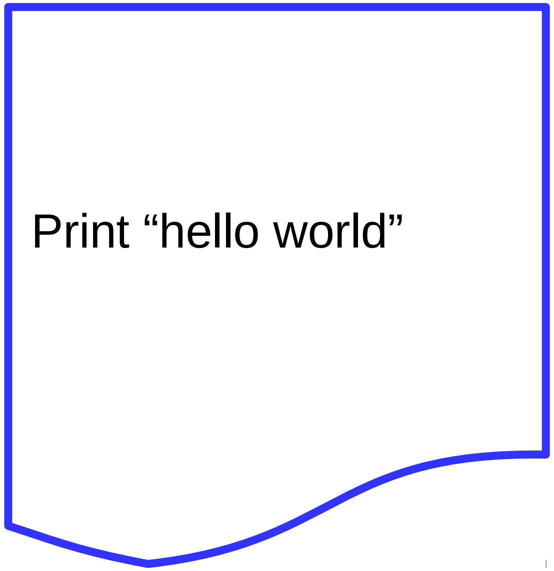
Interactive use

```
$ python
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello, world!'
Hello, world!
>>> 3
$
```

Ctrl-D

Batch mode

example.py



Print “hello world”

```
$ python example.py
Hello world
```

Installing Python

- Python is installed on the PCs.
- Python for Win/Mac/Unix/Linux is available from www.python.org.
  - Generally an easy install.
  - On macs, already part of OS X.

- GUI development environment: IDLE.

Credits:
http://hkn.eecs.berkeley.edu/~dyoo/python/idle_intro/index.html

IDLE Development Environment

- Shell for interactive evaluation.
- Text editor with color-coding and smart indenting for creating python files.
- Menu commands for changing system settings and running files.
- We will use IDLE in class.

Documentation

- Python Documentation
  - http://docs.python.org/2.7/
- Python Qusick reference guide
  - http://rgruet.free.fr/PQR27/PQR2.7.html
- The Python Language Reference
  - http://docs.python.org/2.7/reference/
- The Python Standard Library
  - http://docs.python.org/2.7/library/

Python Tutorials

Things to read through

- “Dive into Python” (Chapters 2 to 4)
  http://diveintopython.org/
- Python 101 – Beginning Python
  http://www.rexx.com/~dkuhlman/python_101/python_101.html

Things to refer to

- The Official Python Tutorial
  http://www.python.org/doc/current/tut/tut.html
- The Python Quick Reference
  http://rgruet.free.fr/PQR2.3.html

Look at a sample of code…

```
x = 34 - 23            # A comment.
y = “Hello”            # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”   # String concat.
print x
print y
```

Look at a sample of code…

```
x = 34 - 23            # A comment.
y = “Hello”            # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”   # String concat.
print x
print y
```

Enough to Understand the Code

- Assignment uses **=**
- comparison uses **==**
- For numbers `+-*/%` are as expected
  - Special use of **+** for string concatenation
  - Special use of **%** for string formatting.
- The basic printing command is “print.”

Enough to Understand the Code

- Logical operators are words (`and, or, not`)
  - *not symbols (`&&, ||, !`).*
- First assignment to a variable will create it.
  - Variable types don’t need to be declared.
  - Python figures out the variable types on its own.
  -
- A variable can
  - Change value
  - change type

Basic Datatypes

- Integers (default for numbers)
  - z = 5 / 2   # Answer is 2, integer division.
- Floats
  - x = 3.456
- Strings
  - Can use “” or ‘’ to specify.   “abc” ‘abc’ (Same thing.)
  - Unmatched ones can occur within the string. “matt’s”
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them: “““a‘b“c”””

Whitespace

- Whitespace is meaningful in Python: especially indentation and placement of newlines.
  - Use a newline to end a line of code.
    (Not a semicolon like in C++ or Java.)
    (Use \ when must go to next line prematurely.)
  - No braces { } to mark blocks of code in Python…
    Use consistent indentation instead. The first line with a new indentation is considered outside of the block.
  - Often a colon appears at the start of a new block.

Comments

- Start comments with # – the rest of line is ignored.
- Can include a “documentation string” as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it’s good style to include one.

```
def my_function(x, y):
  “““This is the docstring. This
function does blah blah blah.”””
# The code would go here...
```

Look at a sample of code…

```
x = 34 - 23            # A comment.
y = “Hello”            # Another one.
z = 3.45
if z == 3.45 or y == “Hello”:
    x = x + 1
    y = y + “ World”   # String concat.
print x
print y
```

Python and Types

- Python determines the data types in a program automatically.
  - Dynamic Typing
- But Python's not casual about types, it enforces them after it figures them out.
  - "Strong Typing"
- So, for example, you can't just append an integer to a string. You must first convert the integer to a string itself.

```
x = "the answer is "   # Decides x is string.
y = 23                 # Decides y is integer.
print x + y    # Python will complain about this.
```

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob   Bob   _bob   _2_bob_   bob_2   BoB
```

- There are some reserved words:

```
 and, assert, break, class, continue, def, del,
elif, else, except, exec, finally, for, from,
global, if, import, in, is, lambda, not, or, pass,
print, raise, return, try, while
```

Accessing Non-existent Name

- If you try to access a name before it’s been properly created (by placing it on the left side of an assignment), you’ll get an error.

```
>>> y

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name ‘y' is not defined
>>> y = 3
>>> y
3
```

Multiple Assignment

- You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

String Operations

- We can use some methods built-in to the string data type to perform some formatting operations on strings:

```
>>> “hello”.upper()
‘HELLO’
```

- There are many other handy string operations available. Check the Python documentation for more.
- str(Object)
  - returns a String representation of the Object

Printing with Python

- You can print a string to the screen using “print.”
- Using the % string operator in combination with the print command, we can format our output text.

```
    >>> print  "%s xyz %d"  %  ("abc", 34)
abc xyz 34
```

- “Print” automatically adds a newline to the end of the string.
- If you include a list of strings, it will concatenate them with a space between them.

```
    >>> print "abc"    >>> print "abc", "def"
    abc    abc def
```

Input

- The raw_input(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods int(string), float(string), etc.

Python2 vs python 3

- Python 2
  - print “abc”
  - `raw_input("> ")`

- Python 3s
  - print ("abc")
  - raw_input("> ")

Input: Example

```
print "What's your name?"
name = raw_input("> ")

print "What year were you born?"
birthyear = int(raw_input("> "))

print "Hi %s! You are %d years old!" %
(name, 2011 - birthyear)
```

Problem

- Implement a program that reads two numbers from the keyboard and calculates their average

Booleans

- 0 and None are false
- Everything else is true
- True and False are aliases for 1 and 0 respectively

Boolean Expressions

- Compound boolean expressions short circuit
- **and** and **or** return one of the elements in the expression
- Note that when None is returned the interpreter does not print anything

```
>>> True and False
False
>>> False or True
True
>>> 7 and 14
14
>>> None and 2
>>> None or 2
2
```

No Braces

- Python uses indentation instead of braces to determine the scope of expressions
- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)
- This forces the programmer to use proper indentation since the indenting is part of the program!

If Statements

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30  :
    y = x + 30
else :
    y = x
print ‘y = ‘,
print math.sin(y)
```

While loops

```
x = 1
while x < 10 :
    print x
    x = x + 1
```

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

The Loop Else Clause

- The optional else clause runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print x
    x = x + 1
else:
    print 'hello'
```

```
1
2
hello
```

The Loop Else Clause

- The optional else clause runs only if the loop exits normally (**not by break**)

```
x = 1

while x < 3 :
    print x
    x = x + 1
else:
    print 'hello'
```

```
1
2
hello
```

For Loops

- Similar to perl for loops, iterating through a list of values

```
for x in [1,7,13,2]:
    print x
```

```
for x in range(5) :
    print x
```

Problem

- Implement a program that reads 20 numbers from the keyboard and calculates their average
  - If the numbers are positive

Files: Input

inflobj = open(‘data’, ‘r’)	Open the file ‘data’ for reading
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

- https://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files

```
f = open(‘data’,‘r’)
s1 = f.read()
print s1
```

```
f = open(‘data’,‘r’)
v1 = f.readlines()
for s in v1
   print s
```

Files: Input

inflobj = open(‘data’, ‘r’)	Open the file ‘data’ for reading
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

- https://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files

```
f = open(‘data’, ‘r’)
for line in f:
        print line,
```

Files: Output

Open the file ‘data’ for writing	Open the file ‘data’ for writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

Exception

- If file does not exist?
  - inflobj = open(‘data’, ‘r’)
    - Traceback (most recent call last):
    -   File "<stdin>", line 1, in <module>
    - IOError: [Errno 2] No such file or directory: '5.cdd'
- Try/except

https://docs.python.org/2/tutorial/errors.html

Try/except

- To catch one exceptions

```
try:
   inflobj = open(‘data)
except IOError:
   print "Oops! That file does not exist..."
```

- To catch many exceptions

```
Try:
    ...
except (RuntimeError, TypeError, NameError):
    ...
```

- To catch all exceptions

```
try:
   …..
except :
   print "Oops!"
```

- To catch many exceptions

```
Try
   …
except IOError as e:
   …
except ValueError:
   …
except:
   …..
```

Try/except/else

```
try:
....
except IOError:
....
else:
....
```

- Executed when exception not raised

Try/except/else

```
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'
```

Problem

- Implement a program that reads a file containing one number per line
  - Prints the values on the screen
-

Lists

- Ordered collection of data
- Data can be of different types
- Lists are mutable
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists: Modifying Content

- `x[i] = a`
  - reassigns the ith element to the value a
- Since x and y point to the same list object, both are changed
- The method append also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Tuples

- Tuples are immutable versions of lists
- One strange point is the format to make a tuple with one element:
- ‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Substrings and Methods

- len(String)
  - returns the number of characters in the String
- >>> s = '012345678'
- >>> s[3]
- '3'
- >>> s[1:4]
- '123'
- >>> s[1:4:2]
- '13'

```
>>> s = '012345678'
>>>
>>> s[2:]
'2345678'
>>> s[:4]
'0123'
>>> s[-2]
'4'
>>> s[::-1]
'876543210'
```

•Substrings and Methods

```
>>> s = '012345'
>>> s[3]
'3'
>>> s[1:4]
'123'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-2]
'4'
```

- **len**(String) – returns the number of characters in the String

- **str**(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>>
str(10.3)
'10.3'
```

Problem

- Implement a program that read a file containing one number per line
  - Stores the values on a array
-

Dictionaries

- A set of key-value pairs
- Dictionaries are mutable
- { }
  - Empty dictionary

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' :
[1,2,3]}

>>> d

{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}

>>> d['blah']

[1, 2, 3]
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99,
'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The del method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

Problem

- Implement a program that read a file containing one number per line
  - Counts the ocurrence of each value
-

Function Basics

```
def max(x,y) :
   if x < y :
      return x
   else :
      return y
def div(x, y):
   Return x/y,  x%y
```

```
>>> max(5, 3)
5
>>> div(17,5)
(3, 2)
>>> ret = div(17,5)
>>>ret[1]
2
>>> ret1, ret2 = div(17,5)
>>> ret1
3
```

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the def statement simply assigns a function to a variable

```
>>> func = div
>>> func(12,3)
(4, 0)
```

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print 'hello'
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Functions as Parameters

```
def foo(f, a) :
    return f(a)
def bar(x) :
    return x * x
```

```
>>> foo(bar, 3)
9
```

- The function **foo** takes two parameters
  - applies the first as a function with the second as its parameter

Higher-Order Functions

```
def double(x):
    return 2*x
```

```
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> map(double,lst)
[0,2,4,6,8,10,12,14,16,18]
```

- map(func,seq)
  - for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
for x in seq:
   new_seq.append(func(x))
```

Higher-Order Functions

```
def even(x):
    return(x%2) == 0
```

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

- filter(boolfunc,seq)
  - returns a sequence containing all those items in **seq** for which **boolfunc** is True.

```
for x in seq:
   If boolfunc(x):
      new_seq.append(x)
```

Higher-Order Functions

```
def plus(x,y):
    return (x + y)
```

```
>>> lst = range(6)
>>> lst
[0, 1, 2, 3, 4, 5]
>>> reduce(plus, lst)
15
```

- reduce(func,seq)
  - applies **func** to the items of **seq**, from left to right, two-at-time, to reduce the **seq** to a single value.

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn’t limit the type of a parameter

```
>>> def foo(x = 3) :
...     print x
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```