

## Section I

# Python Introduction

# What is Python?

- An interpreted high-level programming language.
- Similar to Perl, Ruby, Tcl, and other so-called "scripting languages."
- Created by Guido van Rossum around 1990. Named in honor of Monty Python

# Where to Get Python?

<http://www.python.org>

<http://continuum.io/downloads.html>

- Downloads
- Documentation and tutorial Community Links
- Third party packages
- News and more

# Python Versions

- Version 2.X (most common)
  - Most users use "CPython"
  - Version 3.X (bleeding edge, the future) Alternative implementations
- Jython
- IronPython
- PyPy

# Some Uses of Python

- Text processing/data processing Application scripting
- Systems administration/programming Internet programming
- Graphical user interfaces
- Testing
- Writing quick "throw-away" code

# Running Python

- Python programs run inside an interpreter
- The interpreter is a simple "console-based" application that normally starts from a command shell (e.g., the Unix shell)

```
$python
```

```
Python 2.7.5 |Anaconda 1.5.1 (x86_64)| (default, May 31 2013, 10:42:42)
```

```
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
```

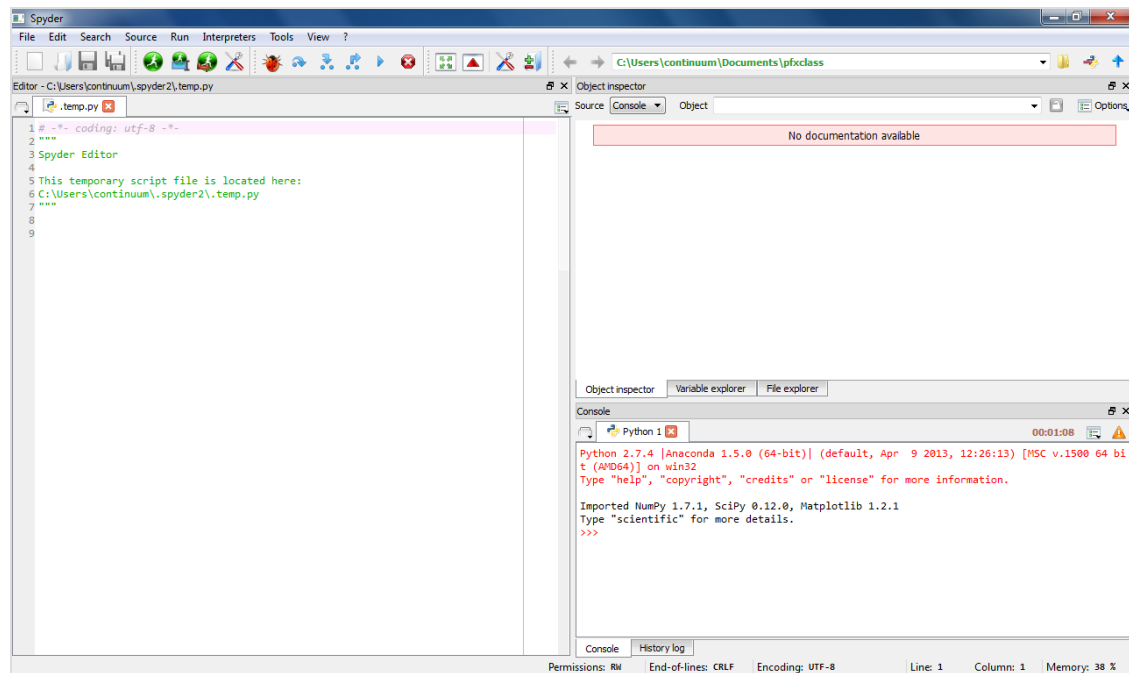
```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

- Expert programmers usually have no problem using the interpreter in this way, but it's not so user-friendly for beginners

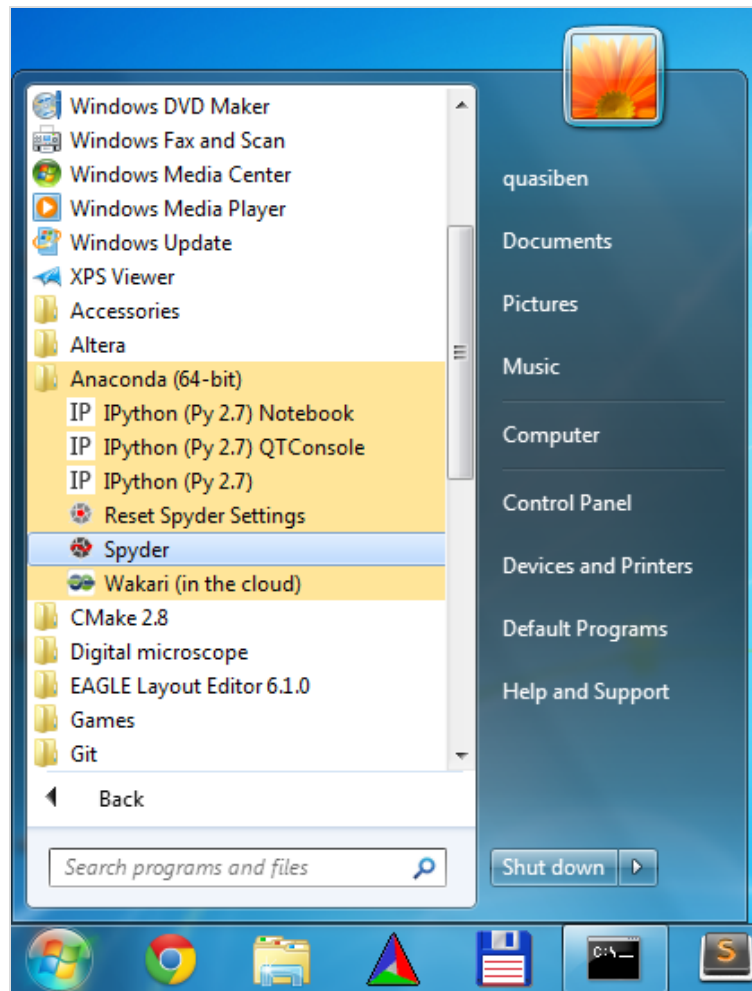
# Spyder

- Integrated Development Environment
- Runs on all platforms
- Built-in console, IPython, and debugger
- Many Additional Features



# Spyder on Windows

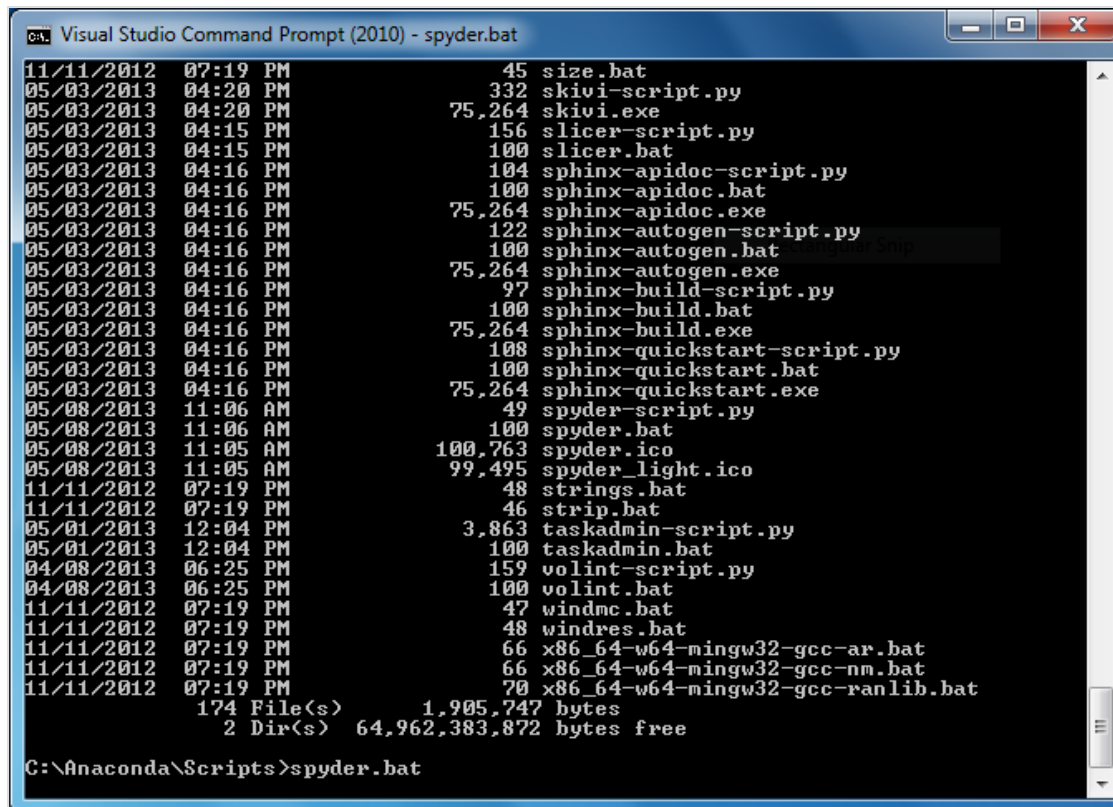
- Look for it in the "Start" menu





# Running Spyder (Windows)

- Run Spyder from the command line



```
Visual Studio Command Prompt (2010) - spyder.bat
11/11/2012 07:19 PM          45 size.bat
05/03/2013 04:20 PM        332 skivi-script.py
05/03/2013 04:20 PM       75,264 skivi.exe
05/03/2013 04:15 PM        156 slicer-script.py
05/03/2013 04:15 PM        100 slicer.bat
05/03/2013 04:16 PM        104 sphinx-apidoc-script.py
05/03/2013 04:16 PM        100 sphinx-apidoc.bat
05/03/2013 04:16 PM       75,264 sphinx-apidoc.exe
05/03/2013 04:16 PM        122 sphinx-autogen-script.py
05/03/2013 04:16 PM        100 sphinx-autogen.bat
05/03/2013 04:16 PM       75,264 sphinx-autogen.exe
05/03/2013 04:16 PM         97 sphinx-build-script.py
05/03/2013 04:16 PM        100 sphinx-build.bat
05/03/2013 04:16 PM       75,264 sphinx-build.exe
05/03/2013 04:16 PM        108 sphinx-quickstart-script.py
05/03/2013 04:16 PM        100 sphinx-quickstart.bat
05/03/2013 04:16 PM       75,264 sphinx-quickstart.exe
05/08/2013 11:06 AM         49 spyder-script.py
05/08/2013 11:06 AM        100 spyder.bat
05/08/2013 11:05 AM      100,763 spyder.ico
05/08/2013 11:05 AM      99,495 spyder_light.ico
11/11/2012 07:19 PM         48 strings.bat
11/11/2012 07:19 PM         46 strip.bat
05/01/2013 12:04 PM       3,863 taskadmin-script.py
05/01/2013 12:04 PM        100 taskadmin.bat
04/08/2013 06:25 PM        159 volint-script.py
04/08/2013 06:25 PM        100 volint.bat
11/11/2012 07:19 PM         47 windmc.bat
11/11/2012 07:19 PM         48 windres.bat
11/11/2012 07:19 PM        66 x86_64-w64-mingw32-gcc-ar.bat
11/11/2012 07:19 PM        66 x86_64-w64-mingw32-gcc-nm.bat
11/11/2012 07:19 PM        70 x86_64-w64-mingw32-gcc-ranlib.bat
174 File(s)          1,905,747 bytes
2 Dir(s)          64,962,383,872 bytes free

C:\Anaconda\Scripts>spyder.bat
```

- C:\Anaconda\Scripts>spyder.bat

# Running on other Systems

- Launch a terminal or command shell
- `$ spyder`

# The Python Interpreter

- When you start Python, you get an "interactive" mode where you can experiment
- If you start typing statements, they will run immediately
- No edit/compile/run/debug cycle
- In fact, there is no "compiler"

# Interactive Mode

- The interpreter runs a "read-eval" loop

```
>>> print "hello world"
hello world
>>> 37*42
1554
>>> for i in range(5):
...     print i
...
0
1
2
3
4
>>>
```

- Very useful for debugging, exploration

# Interactive Mode

- Some notes on using the interactive shell
  - `>>>` is the interpreter prompt for starting a new statement
  - `...` is the interpreter prompt for continuing a statement (it may be blank in some tools)

# Interactive Mode

- Use underscore (\_) for the last result

```
>>> 37*42 1554
>>> _ * 2 3108
>>> _ + 50 3158
>>>
```

- Note: This only works in interactive mode (you never use \_ in a program)

# Getting Help

- *help(name) command*

```
>>> help(range)
Help on built-in function range in module __builtin__:
    range(...)
        range([start,] stop[, step]) -> list of integers
        Return a list containing an arithmetic progression of
        range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!)
        ...
>>>
```

- Type `help()` with no name for interactive help
- Documentation at <http://docs.python.org>

# Creating Programs

- Programs are saved in .py files

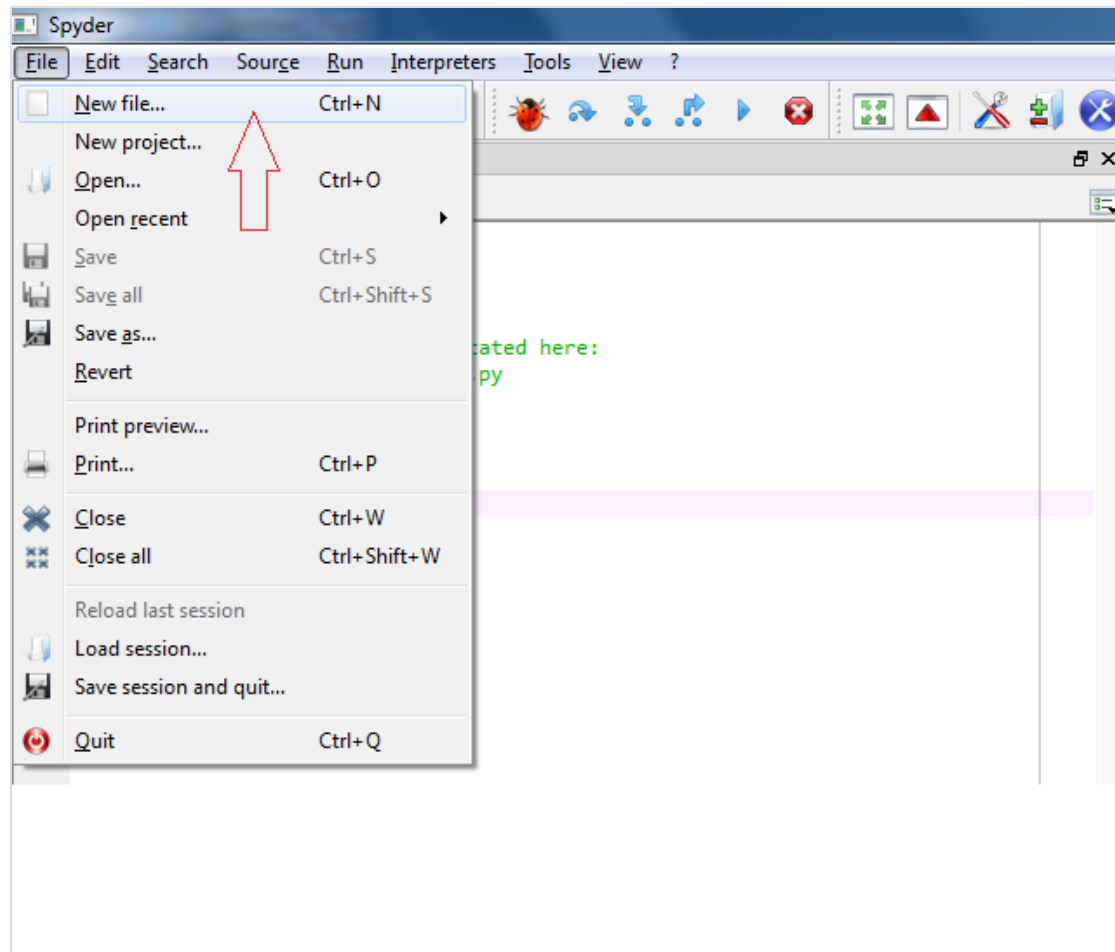
```
# helloworld.py  
print "hello world"
```

- Source files are simple text files
- Create with your favorite editor (e.g., emacs)
- Can also edit programs with IDLE or other Python IDE (too many to list)



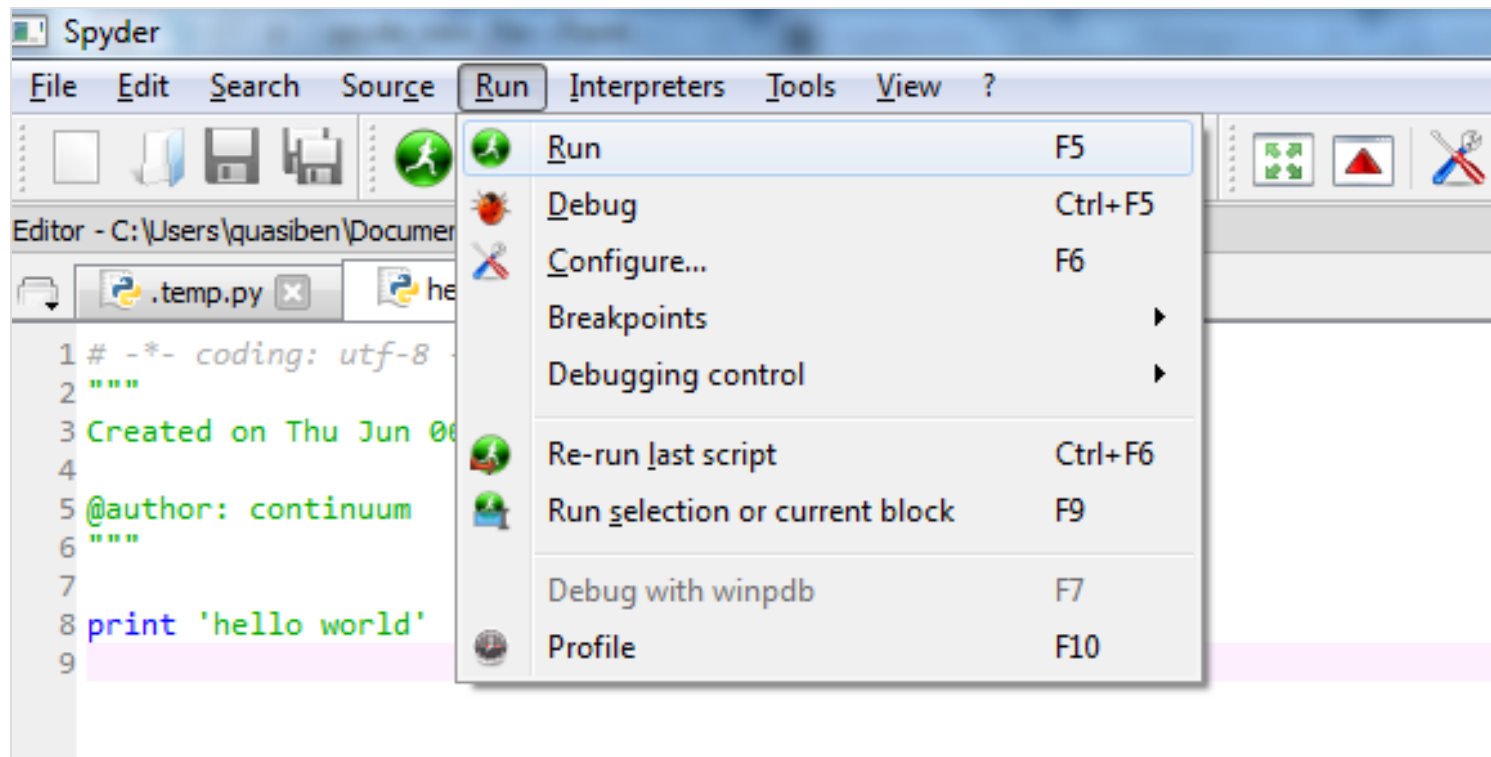
# Creating Programs

- Creating a new program in Spyder



# Running Programs

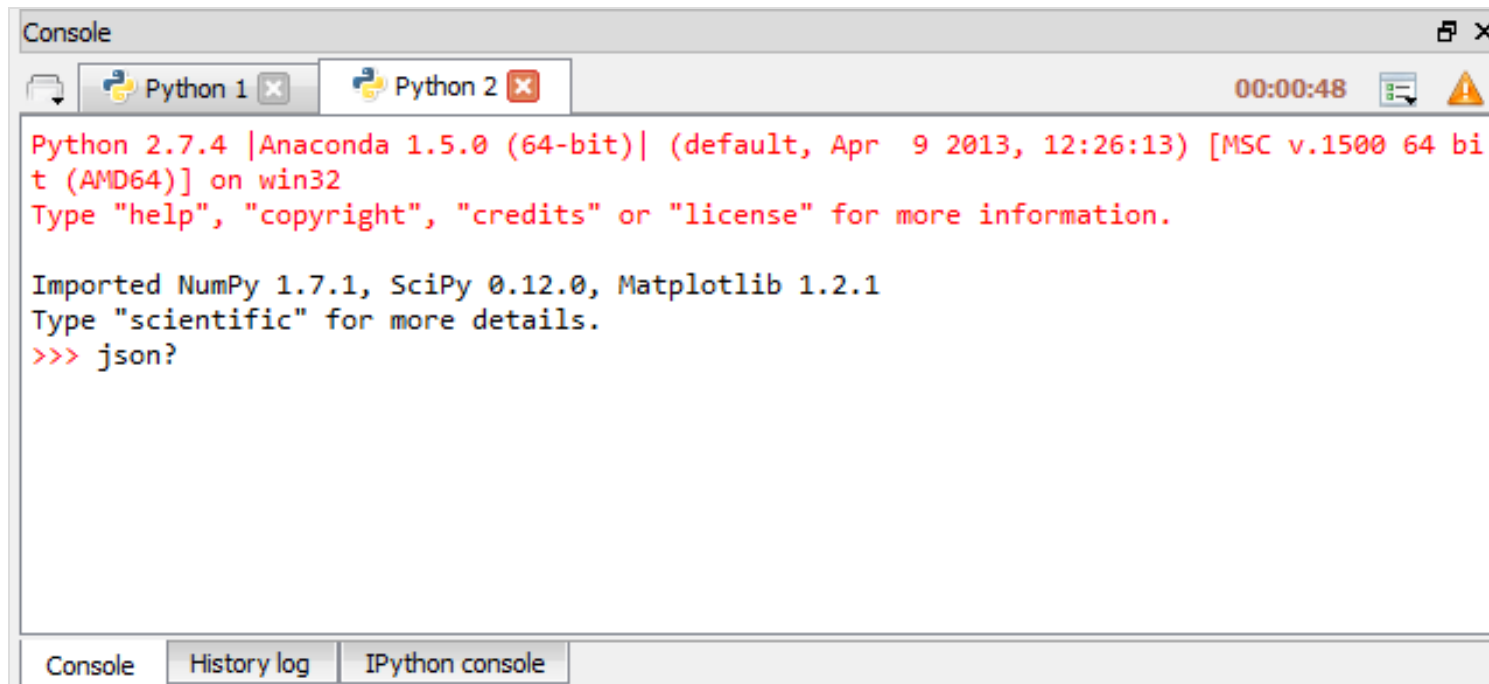
- Running programs **Run Menu** or **F5**



- Will see output in Console Inspector

# Additional Features

- Tabbed Consoles



# Running Programs (CLI)

- In production environments, Python may be run from command line or a script
- Command line (Unix)

```
$python helloworld.py  
hello world  
$
```

- Command shell (Windows)

```
C:\Documents\pfxclass>python.exe helloworld.py  
hello world
```

```
C:\Documents\pfxclass>python.exe helloworld.py  
hello world
```

# A Sample Program

- The Sears Tower Problem

One morning, you go out and place a dollar bill on the sidewalk by the Sears tower. Each day thereafter, you go out double the number of bills. How long does it take for the stack of bills to exceed the height of the tower?

# A Sample Program

```
# sears.py
bill_thickness = 0.11 * 0.001    # Meters (0.11 mm)
sears_height    = 442 # Height (meters)
num_bills      = 1
day            = 1

while num_bills * bill_thickness < sears_height:
    print day, num_bills, num_bills * bill_thickness
    day = day + 1
    num_bills = num_bills * 2

print "Number of days", day
print "Number of bills", num_bills
print "Final height", num_bills * bill_thickness
```

# A Sample Program

- Output

```
#python sears.py
```

```
1 1 0.00011
2 2 0.00022
3 4 0.00044
4 8 0.00088
5 16 0.00176
6 32 0.00352
```

```
...
```

```
21 1048576 115.34336
```

```
22 2097152 230.68672
```

```
Number of days 23
```

```
Number of bills 4194304
```

```
Final height 461.37344
```

# Python 101 : Statements

- A Python program is a sequence of statements
- Each statement is terminated by a newline
- Statements are executed one after the other until you reach the end of the file.
- When there are no more statements, the program stops



# Python 101 : Comments

- Comments are denoted by #

```
# This is a comment  
height      = 442           # Meters
```

- Extend to the end of the line
- There are no block comments in Python (e.g., /\* ... \*/).

# Python 101 : Variables

- A variable is just a name for some value
- Variable names follow same rules as C [A-Za-z\_][A-Za-z0-9\_]\*
- You do not declare types (int, float, etc.)

```
height = 442           # An integer
height = 442.0         # Floating point
height = "Really tall" # A string
```

- Differs from C++/Java where variables have a fixed type that must be declared.

# Python 101 : Keywords

- A variable is just a name for some value
- Variable names follow same rules as C  
`[A-Za-z_][A-Za-z0-9_]*`
- You do not declare types (int, float, etc.)

<code>and</code>	<code>elif</code>	<code>if</code>	<code>print</code>
<code>as</code>	<code>else</code>	<code>import</code>	<code>raise</code>
<code>assert</code>	<code>except</code>	<code>in</code>	<code>return</code>
<code>break</code>	<code>exec</code>	<code>is</code>	<code>try</code>
<code>class</code>	<code>finally</code>	<code>lambda</code>	<code>while</code>
<code>continue</code>	<code>for</code>	<code>not</code>	<code>with</code>
<code>def</code>	<code>from</code>	<code>or</code>	<code>yield</code>
<code>del</code>	<code>global</code>	<code>pass</code>	

- Variables can not have one of these names
- These are mostly C-like and have the same meaning in most cases (later)

# Python 101 : Looping

- The **while** statement executes a loop
- These are all different variables:

```
while num_bills * bill_thickness < sears_height:  
    print day, num_bills, num_bills * bill_thickness  
    day = days + 1  
    num_bills = num_bills * 2
```

- Executes the indented statements underneath while the condition is true

# Python 101 : Indentation

- Indentation used to denote blocks of code
- Indentation must be consistent

```
while num_bills * bill_thickness < sears_height:  
    print day, num_bills, num_bills * bill_thickness  
    day = days + 1 # ERROR  
    num_bills = num_bills * 2
```

- Colon (:) indicates the start of a block

```
while num_bills * bill_thickness < sears_height:
```

# Python 101 : Indentation

- There is a preferred indentation style
  - Always use spaces
  - Use 4 spaces per level
  - Avoid tabs
- Always use a Python-aware editor

# Python 101 : Conditionals

- If-else

```
if a < b:  
    print "Computer says no"  
else:  
    print "Computer says yes"
```

- If-elif-else

```
if a == '+':  
    op = PLUS  
elif a == '-':  
    op = MINUS  
elif a == '*':  
    op = TIMES  
else:  
    op = UNKNOWN
```

# Python 101 : Relations

- Relational Operators

< > <= >= == !=

- Boolean Expression (and, or, not)

```
if b >= a and b <= c:  
    print "b is between a and c"  
if not (b < a or b > c):  
    print "b is still between a and c"
```



# Python 101 : Truth Values

- Evaluates as "True"
  - Non-zero numbers
  - Non-empty strings
  - Non-empty containers (lists, dicts, etc.)
- Evaluates as "False"
  - 0 (Zero)
  - Empty string or containers

# Python 101 : Printing

- The print statement

```
print x
print x,y,z
print "Your name is", name
print x,                # Omits newline
```

- Produces a single line of text
- Items are separated by spaces
- Always prints a newline unless a trailing comma is added after last item

# Python 101 : User Input

- To read a line of typed user-input

```
name = raw_input("Enter you name:")
```

- Prints a prompt, returns the typed response
- This might be useful for small programs or for simple debugging
- It is not widely used for real programs (we're rarely going to use it in this class)

# Python 101 : pass statement

- Sometimes you will need to specify an empty block of code (like {} in C/Java)

```
if name in namelist:  
    #Not implemented yet (or nothing)  
    pass  
else:  
    statements
```

- **pass** is a "no-op" statement
- It does nothing, but serves as a placeholder for statements (possibly to be added later)

# Python 101 : Long Lines

- Sometimes you get long statements that you want to break across multiple lines
- Use the line continuation character (\)

```
if product=="game" and type=="pirate memory" \  
    and age >= 4 and age <= 8:  
    print "I'll take it!"
```

- However, not needed for code in (),[], or {}

```
if (product=="game" and type=="pirate memory"  
    and age >= 4 and age <= 8):  
    print "I'll take it!"
```

# Basic Datatypes

- Python only has a few primitive types of data
- Numbers
- Strings (character text)

# Numbers

- Python has 4 types of numbers
  - Booleans
  - Integers
  - Floating point
  - Complex (imaginary numbers)

# Booleans

- Two values: True, False

```
a = True  
b = False
```

- Evaluated as integers with value 1,0

```
c = 4 + True      # c = 5  
d = False  
if d == 0:  
    print "d is False"
```

- Although doing that in practice would be odd



# Integers

- Signed values of arbitrary size

```
a = 37
b = -299392993727716627377128481812241231
c = 0x7fa8          # Hexadecimal
d = 0o253           # Octal
e = 0b10001111      # Binary
```

- There are two internal representations
  - int : Small values (less than 32-bits in size)
  - long : Large values (arbitrary size)
- Sometimes see 'L' shown on end of large values

```
>>> b
-299392993727716627377128481812241231L
```

# Integer Operations

+	Add
-	Subtract
*	Multiply
/	Divide
//	Floor divide
%	Modulo
**	Power
<<	Bit shift left
>>	Bit shift right
&	Bit-wise AND
	Bit-wise OR
^	Bit-wise XOR
~	Bit-wise NOT
abs(x)	Absolute value
pow(x,y[,z])	Power with optional modulo (x**y)%z
divmod(x,y)	Division with remainder

# Integer Division

- Classic division (/) - truncates

```
>>> 5/4
1
>>>
```

- Floor division (//) - truncates (same)

```
>>> 5//4
1
>>>
```

- Future division (/) - Converts to float

```
from __future__ import division
5/4
1.25
```

- In Python 3, / always produces a float
- If truncation is intended, use //

# Floating point (float)

- Use a decimal or exponential notation

```
a = 37.45  
b = 4e5  
c = -1.345e-10
```

- Represented as double precision using the native CPU representation (IEEE 754)

```
17 digits of precision  
Exponent from -308 to 308
```

- Same as the C double

# Floating Point Operators

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo (remainder)
**	Power
pow(x,y [,z])	Power modulo (x**y)%z
abs(x)	Absolute value
divmod(x,y)	Division with remainder

- Additional functions are in the math module

```
import math
a = math.sqrt(x)
b = math.sin(x)
c = math.cos(x)
d = math.tan(x)
e = math.log(x)
```

# Converting Numbers

- Type Name can be used to convert

```
a = int(x)           #Convert x to integer
b = float(x)         #Convert x to float
```

- Example:

```
>>> a = 3.14159
>>> int(a)
3
>>>
```

- Also work with string containing numbers

```
>>> a = "3.14159"
>>> float(a)
3.1415989999999999
>>> int("0xff",16)      #Optional integer base
255
```

# Strings

- Written in programs with quotes

```
a = "Yeah but no but yeah but..."
b = 'computer says no'
c = '''
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes,
not around the eyes,
don't look around the eyes,
look into my eyes, you're under.
'''
```

- Standard escape characters work (e.g., '\n')
- Triple quotes captures all literal text enclosed

# String Escape Codes

- In quotes, standard escape codes work

<code>'\n'</code>	Line feed
<code>'\r'</code>	Carriage return
<code>'\t'</code>	Tab
<code>'\xhh'</code>	Hexadecimal value
<code>'\"'</code>	Literal quote
<code>'\\'</code>	Backslash

- The codes are inspired by C



# String Representation

- An ordered sequence of bytes (characters)
- Stores 8-bit data (ASCII)
- May contain binary data, control characters, etc.
- Strings are frequently used for both text and for raw-data of any kind

# String Representation

- Strings work like an array :  $s[n]$

```
a = "Hello world"
1- 59
b = a[0]
c = a[4]
d = a[-1]
# b = 'H'
# c = 'o'
# d = 'd' (Taken from end of string)
```

- Slicing/substrings :  $s[start:end]$

```
d = a[:5] # d = "Hello"
e = a[6:] # e = "world"
f=a[3:8] #f="lowo"
g = a[-5:] # g = "world"
```

- Concatenation (+)

```
a = "Hello" + "World"
b = "Say " + a
```

# More String Operations

- Length (len)

```
>>> s = "Hello" >>> len(s)
5
>>>
```

- Membership test (in)

```
>>> 'e' in s True
>>> 'x' in s False
>>> "ello" in s True
```

- Replication (s\*n)

```
>>> s = "Hello"
>>> s*5
'HelloHelloHelloHelloHello'
>>>
```

# String Methods

- Strings have "methods" that perform various operations with the string data.
- Stripping any leading/trailing whitespace

```
t = s.strip()
```

- Case conversion

```
t = s.lower()  
t = s.upper()
```

- Replacing text

```
t = s.replace("Hello", "Hallo")
```

# More String Methods

<code>s.endswith(suffix)</code>	<code># Check if string ends with suffix</code>
<code>s.find(t)</code>	<code># First occurrence of t in s</code>
<code>s.index(t)</code>	<code># First occurrence of t in s</code>
<code>s.isalpha()</code>	<code># Check if characters are alphabetic</code>
<code>s.isdigit()</code>	<code># Check if characters are numeric</code>
<code>s.islower()</code>	<code># Check if characters are lower-case</code>
<code>s.isupper()</code>	<code># Check if characters are upper-case</code>
<code>s.join(slist)</code>	<code># Joins lists using s as delimiter</code>
<code>s.lower()</code>	<code># Convert to lower case</code>
<code>s.replace(old,new)</code>	<code># Replace text</code>
<code>s.rfind(t)</code>	<code># Search for t from end of string</code>
<code>s.rindex(t)</code>	<code># Search for t from end of string</code>
<code>s.split([delim])</code>	<code># Split string into list of substrings</code>
<code>s.startswith(prefix)</code>	<code># Check if string starts with prefix</code>
<code>s.strip()</code>	<code># Strip leading/trailing space</code>
<code>s.upper()</code>	<code># convert to upper case</code>

- Consult a reference for gory details

# Using dir()

- `dir(obj)` will list available methods/attributes

```
>>> s = "Hello World"
>>> dir(s)
['__add__', '__class__', ..., 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', ... 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

- Use `help()` for more information

```
>>> help(s.zfill)
Help on built-in function zfill:

zfill(...)
    S.zfill(width) -> string
```

# String Mutability

- String are "immutable" (read only)
- Once created, the value can't be changed

```
>>> s = "Hello World"
>>> s[1] = 'a'
Traceback (most recent call last):
  File "", line 1, in
TypeError: 'str' object does not support item assignment
>>>
```

- All operations and methods that manipulate string data always create new strings

# String Conversions

- Use `str()` to convert to a string

```
s = str(obj)
```

- Produces the same text as `print`
- Actually, `print` uses `str()` for output

```
>>> x = 42 >>> str(x) '42'  
>>>
```



# Special Strings

- String literals are sometimes prefixed by a special code that affect semantics

```
u'Jalape\u00f1o'      # Unicode (international chars)
b'\x12\x27@\x00'      # Bytes
r'c:\newdata\test'    # Raw (uninterpreted backslash)
```

- Unicode strings store multi-byte characters
- Raw strings leave backslashes intact
- 8-bit bytes only allowed in byte strings

# String Splitting

- Strings often represent fields of data
- To work with each field, split into a list

```
>>> line = 'GOOG 100 490.10'  
>>> fields = line.split()  
>>> fields  
['GOOG', '100', '490.10'] >>>
```

- Example: When reading data from a file, you might read each line and then split the line into columns.

# Lists

- Array of arbitrary values

```
names = [ "Elwood", "Jake", "Curtis" ]  
nums  = [ 39, 38, 42, 65, 111]
```

- Can contain mixed data types

```
items = [ "Elwood", 39, 1.5 ]
```

- Adding new items (append, insert)

```
items.append("that")    # Adds at end  
items.insert(2,"this")  # Inserts in middle
```

- Concatenation : s + t

```
s = [1,2,3]  
t = ['a','b']  
s + t    →    [1,2,3,'a','b']
```

# Lists (cont)

- Lists are indexed by integers (starting at 0)

```
names = [ "Elwood", "Jake", "Curtis" ]  
names[0] → "Elwood"  
names[1] → "Jake"  
names[2] → "Curtis"
```

- Negative indices are from the end

```
names[-1] → "Curtis"
```

- Changing one of the items

# More List Operations

- Length (len)

```
>>> s = ['Elwood', 'Jake', 'Curtis'] >>> len(s)
3
>>>
```

- Membership test (in)

```
>>> 'Elwood' in s True
>>> 'Britney' in s False
>>>
```

- Replication (s\*n)

```
>>> s = [1,2,3]
>>> s*3
[1,2,3,1,2,3,1,2,3]
>>>
```

# List Removal

- Removing an item

```
names.remove("Curtis")
```

- Deleting an item by index

```
del names[2]
```

- Removal results in items moving down to fill the space vacated (i.e., no "holes").

# List Iteration

- Iterating over the list contents

```
for name in names:  
    # use name  
    ...
```

- Similar to a 'foreach' statement from other programming languages

# List Sorting

- Lists can be sorted "in-place" (sort method)

```
s = [10,1,7,3]
s.sort()          # s = [1,3,7,10]
```

- Sorting in reverse order

```
s = [10,1,7,3]
s.sort(reverse=True) # s = [10,7,3,1]
```

- Sorting works with any ordered type

```
s = ["foo", "bar", "spam"]
s.sort()          # s = ["bar", "foo", "spam"]
```



# Lists and Math

- Caution : Lists weren't designed for "math"

```
>>> nums = [1,2,3,4,5]
>>> nums * 2
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> nums + [10,11,12,13,14]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14] >>>
```

- They don't represent vectors/matrices
- Not the same as in MATLAB, Octave, IDL, etc.
- There are some add-ons for this (e.g., numpy)

# File Input and Output

- Opening a file

```
f = open("foo.txt", "r")    # Open for reading  
g = open("bar.txt", "w")    # Open for writing
```

- To read data

```
data = f.read([maxbytes])
```

 To write text to a file

- To write text to a file

```
g.write("some text")
```

- To close when you're done

```
f.close()
```

# Reading File Data

- Reading an entire file all at once as a string

```
f = open(filename, "r")
data = f.read()
f.close()
```

- Reading an entire text-file line-by-line

```
f = open(filename, "r")
for line in f:
    # Process the line
f.close()
```

# Reading File Data

- End-of-file indicated by an empty string

```
data = f.read(nbytes)
if data == '':
    # No data read.  EOF
    ...
```

- Example: Reading a file in fixed-size chunks

```
f = open(filename, "r")
while True:
    chunk = f.read(chunksize)
    if chunk == '':
        break
    # Process the chunk
    ...
f.close()
```

# Writing File Data

- Writing string data

```
f = open("outfile","w")
f.write("Hello World\n")
...
f.close()
```

- Redirecting the print statement

```
f = open("outfile","w")
print >>f, "Hello World"
...
f.close()
```

# File Management

- Files should be properly closed when done

```
f = open(filename,"r") # Use the file f
...
f.close()
```

- In modern Python (2.6 or newer), use "with"

```
with open(filename,"r") as f:
    # Use the file f
    ...
    statements
```

- This automatically closes the file when control leaves the indented code block

# Type Conversion

- In Python, you must be careful about converting data to an appropriate type

```
x = '37'          # Strings
y = '42'
z = x + y         # z = '3742' (concatenation)
```

```
x = 37
y = 42
z=x+y             #z=79 (integer+)
```

- Differs from Perl/PHP where "+" is assumed to be numeric arithmetic (even on strings)

```
$x = '37';
$y = '42';
$z=$x+$y;         # $z=79
```

# Simple Functions

- Use functions for code you want to reuse

```
def sumcount(n):  
    '''Returns the sum of the first n integers'''  
    total = 0  
    while n > 0:  
        total += n  
        n -= 1  
    return total
```

- Calling a function

```
a = sumcount(100)
```

- A function is just a series of statements that perform some task and return a result



# Library Functions

- Python comes with a large standard library
- Library modules accessed using import

```
import math
x = math.sqrt(10)
import urllib
u = urllib.urlopen("http://www.python.org/index.html")
data = u.read()
```

- Will cover in more detail later

# dir() function

- dir(module) returns all names in a library

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_current_frames', '_getframe', 'api_version',
 'argv', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'exc_clear', 'exc_info', 'exc_type',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'getcheckinterval',
 ...,
 'version_info', 'warnoptions']
```

- Useful for exploring library contents

# Exception Handling

- Errors are reported as exceptions
- An exception causes the program to stop

```
>>> int("N/A")
Traceback (most recent call last):
  File "", line 1, in
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

- For debugging, message describes what happened, where the error occurred, along with a traceback.

# Exceptions

- Exceptions can be caught and handled
- To catch, use try-except statement

```
for line in f:
    fields = line.split()
    try:
        shares = int(fields[1])

    except ValueError:
        print "Couldn't parse", line
    ...
```

Name must match the kind of error you're trying to catch

```
>>> int("N/A")
Traceback (most recent call last):
  File "", line 1, in
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

# Exceptions

- To raise an exception, use the raise statement

```
raise RuntimeError("What a kerfuffle")
```

- Will cause the program to abort with an exception traceback (unless caught by try- except)

```
$python foo.py
Traceback (most recent call last):
  File "foo.py", line 21, in
    raise RuntimeError("What a kerfuffle")
RuntimeError: What a kerfuffle
```

# Summary

- This has been an overview of simple Python
- Enough to write basic programs
- Just have to know the core datatypes and a few basics (loops, conditions, etc.)