Introduction to Python
Lecture I

Worcester Polytechnic Institute
Academic & Research Computing

Introduction

This tutorial is heavily based on the book "Learning Python" by Mark Lutz

However, there are (probably) hundreds of online resources available to practice and learn Python.

Some examples are:

Python Tutorial:

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

Learn Python:

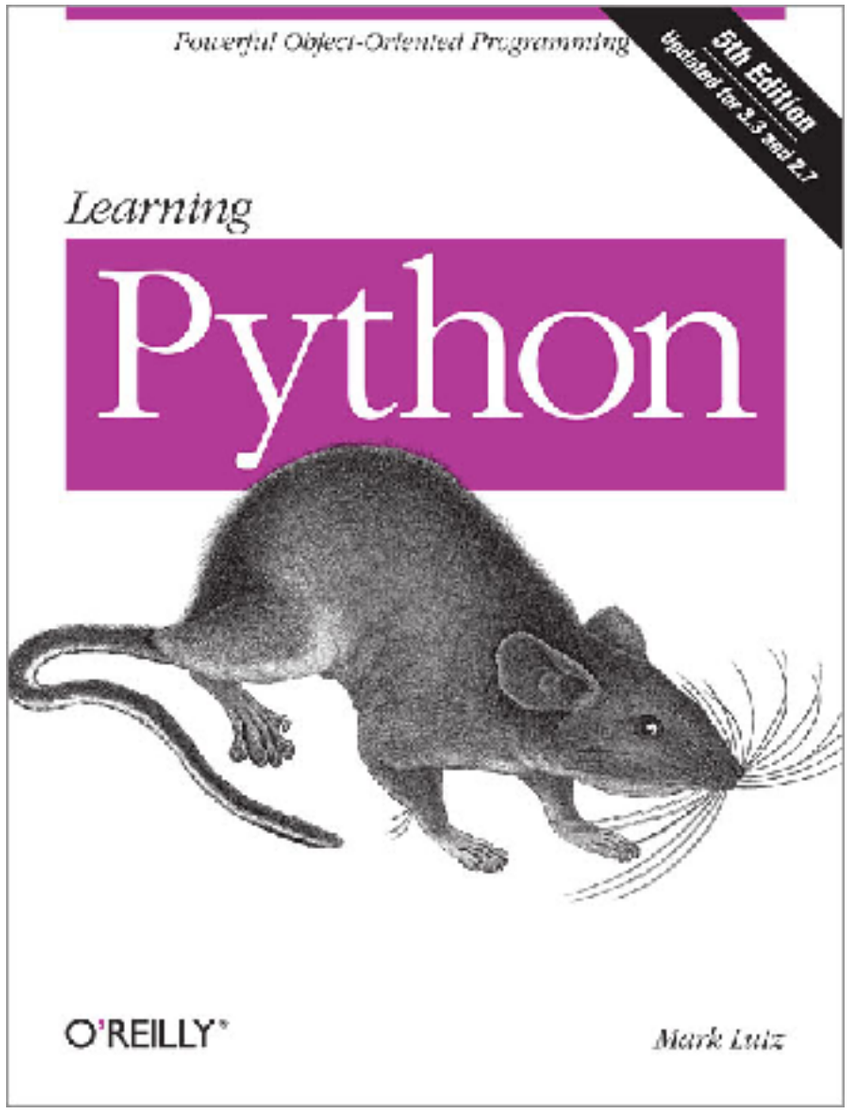
http://www.learnpython.org

Codecademy:

https://www.codecademy.com

Learn Python the Hard Way:

https://learnpythonthehardway.org



The best resources for when you have trouble are:

Google

Stackoverflow (which is where Google usually takes you)

If you are having a problem, odds are someone else has already dealt with it.

Introduction

What is Python?

- High level programming language
- Interpreted, *not* compiled, language
- Emphasizes readability
- Dynamically typed (no need to declare variable types)

What is Python good at?

- Fast development and debugging
- Portability
- Data analysis
- Web development
- Glue language (front-end for C++, Fortran, Java)

What can I use Python for?

- Automate tasks with Python scripts
- Write small programs for data analysis
- Many programs now offer a Python API, allowing you to invoke part or all of a larger program from a Python program (think a workflow for data processing)

Introduction

What is Python bad at?

- Execution speed is not as fast as corresponding C/C++ or Fortran programs

Why?

This goes back to the lack of compilation. Python code is compiled "Just In Time" by the interpreter when you run it, in contrast to C/C++ code that is compiled *before* you run the program.

In addition, Python compilation is not as "low level", meaning that it does not run as machine code directly on the CPU (C/C++/Fortran are compiled to machine code)

This combination means that code *you write* in Python will likely be slower.

However, if you are calling routines or functions from the Python standard library, these parts of your code will "run at C speed"

Do I care if Python is slower?

- Not really

Overview

What we will cover in this lecture

- Getting started (installation, interactive, scripts, modules)
- Syntax
- Numbers and Strings
- Lists
- Files
- Functions

Getting Started: Installation

Linux and macOS have Python installed by default. Python is NOT installed on Windows by default, and will need to be installed explicitly.

If you are using Windows, or Python is not installed for some reason, Python is free and can be downloaded from:

https://www.python.org/downloads/

An alternative to the standard Python installer is Anaconda:

https://www.continuum.io/downloads

Anaconda has the advantage of having not only the standard Python library installed, but also over 100 of the most commonly used packages preinstalled, such as NumPy, Scipy, pandas, beautifulsoup, matplotlib, etc, etc, etc

However, today we will be using a website based Python development environment.

Open Chrome, and go to https://repl.it or Google repl.it

Getting Started: Interactive Prompt

Try the following:

Create a variable `mascot` and set the value of the variable to '*Gompei*' including the quotes
Print the variable `mascot`

Create two integer variables `a` and `b`, set them to 5 and 10 respectively
Print the value of `a`, then the value of `b`

Create a new variable `c` that has the value of the sum of `a` and `b`
Print the result

Now print the sum of `a` and `b`, but do not create a new variable to do it

Add `mascot` to `a`. What happens?

Print a variable `mascot_new` without assigning a value What happens?

Create a floating point variable `f`, set it to 3.1415
Print the value of `f`

Now print the sum of `a` and `f`. What happens? Now just print the integer part.

Try typing an expression such as `5 + 10` into the prompt. What happens?

Getting Started: Interactive Prompt

From that short exercise, we actually learn a lot!

- Python is readable and straightforward
- We don't need to explicitly `print` variables at the interactive prompt, we can just type the variable name
- Python is dynamically typed.
  - We never needed to tell Python we were using an integer or a float
- Mixed operands (int and float) will convert the integer to a float automatically
- We can perform explicit casts (e.g. str(a) or int(f))
- Variables persist throughout our interactive session
- Python prints readable error messages.
  - Built in debugging messages are actually useful:

```
NameError: name ‘mascot_new' is not defined

TypeError: cannot concatenate 'str' and 'int' objects
```

Syntax

Syntax in Python is fairly straightforward and can be summed up as:

spaces matter

Many other programming languages require semicolons and brackets for newlines and control structures, but do not require proper indentation of code.

Why indent your code? It makes it easier to read in general, and it makes it easier to see the control flow of large chunks of code.

Python forces you to indent your code, because the indents are what make up the logical control structure.

This logical alignment makes the code *much easier* to read.

Syntax

A few general rules

Parentheses are optional:

`if (x < y)` is the same as `if x < y`

End of the line is the end of a statement:

```
x = 1
```

End of indentation is the end of a block:

```
if x > y:
   x = 1
   y = 2
```

Interactive Loop Example:

```
while True:
   reply = raw_input('Enter text:')
   if reply == 'stop': break
   print(reply.upper())
print('Bye!')
```

Create a Python script with the above code and run it using the Python interpreter.

Now modify your Python script to calculate the square of any input number.

Run it using the Python interpreter.

Syntax

```
while True:
    reply = raw_input('Enter a number:')
    if reply == 'stop': break
    print(int(reply)**2)
print('Bye!')
```

What happens when you enter a string instead of a number?

We will return to this problem later.

Core Data Types

Python contains a variety of built-in object types for you to use.

Object Type	Example Literals/Creation
Numbers	`1234, 3.1415, Decimal(), Fraction()`
Strings	`'gompei', "Bob's"`
Lists	`[1, [2, 'three'], 4.5], list(range(10))`
Dictionaries	`{'mascot': 'gompei', 'animal': 'goat'}, dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U'), tuple('goat'), namedtuple`
Files	`open('goats.txt'), open(r'C:\goat.bin', 'wb')`
Sets	`set('abc'), {'a','b','c'}`

Core Data Types

Remember, everything in Python is an object, even things not in that table.

When you run the following:

```
>>> ‘goat’
```

you are technically running a literal expression that generates and returns a new string **object**.

There is specific syntax to make this object (the quotes), just as square brackets will make a *list* object, curly brackets will make a *dictionary* object, etc, etc.

As we saw before though, there are no type declarations (int, float, string), and the types are generally determined in the object-generation expression.

Core Data Types

Once an object is created, all of the related operations are bound to that object forever.

You can only perform string operations on strings, list operations on lists, number operations on numbers (we saw this in our loop example).

This is what is meant by Python being **dynamically typed**.

Python includes a built-in function for checking object types:

```
>>> mascot = ‘gompei’
>>> x = 5
>>> y = 3.1415
>>> type(mascot)
<type ‘str’>
>>> type(x)
<type ‘int’>
>>> type(y)
<type ‘float’>
```

Numbers

Numbers in Python are straightforward, and this **object type** contains the usual things you would expect:

- integers
- floats
- complex
- fixed precision decimals
- rationals
- sets

All of our number objects can be manipulated with the normal mathematical operators (+, -, *, **, /)

```
>>>3.1415 * 2
6.283
```

In addition to the basic mathematical operators, Python includes useful numeric modules (*a package of tools*) that can be imported.

Numbers

One such module is the math module:

```
>>>import math
>>>math.pi
3.141592653589793
>>>math.sqrt(4)
2
```

Another is the random module:

```
>>>import random
>>>random.random()
0.7082048489415967
>>>random.choice([1,2,3,4])
1
```

Numbers

Just like what we saw previously with modules, we can list the attributes of **any** object with `dir()`!

Let's see what attributes a number object has:

```
>>> y = 3.1415
>>> dir(y)
['__abs__', '__add__', '__class__', '__coerce__',
'__delattr__', '__div__', '__divmod__',
…
'as_integer_ratio', 'conjugate', 'fromhex', 'hex',
'imag', 'is_integer', 'real']
>>> y.is_integer()
>>> False
```

Strings

Strings in Python have a much more rich set of uses and behaviors. Strings are the first example of a *sequence* in Python, or a:

"…positionally ordered collection of other objects."

What does this mean?

Strings are sequences that are positionally ordered, each item stored left to right, and can be fetched by their positions.

Technically, strings are sequences of single character strings.

Strings

Some examples of string manipulation:

```
>>> G = ‘gompei’
>>> len(G)
6
>>> G[0]
‘G’
>>> G[1]
‘o’
```

Strings

Indexes for strings (and later lists, dictionaries, tuples, etc) are coded as offsets starting at zero.

We can index from the end of the sequence as well:

```
>>> G[-1]
‘i’
>>> G[-2]
‘e’
```

In Python, positive indices count from the left, negative indices count from the right.

Strings

A negative index is simply added to the length of the string, so we can also do:

```
>>> G[len(G) - 1]
'i'
```

You can imagine that we can use an arbitrary expression as our index, so this can be hard coded, but also a variable or an expression as well.

Sequences also support ranges:

```
>>> G[1:4]
'ompe'
```

The notation `G[I:J]` means “everything in G from I up to but not including J”

Strings

For slices, the left bound defaults to zero,

```
>>> G[:4]
‘Gomp’
>>> G[0:4]
‘Gomp’
```

and the right bound will default to the last entry,

```
>>> G
‘Gompei’
>>> G[:]
‘Gompei’
```

Strings

Strings, and more generally *sequences*, support *concatenation* using the plus sign, and repetition:

```
>>> G + ‘Bot’
‘GompeiBot’
>>> G[:4] + ‘Bot’
‘GompBot’
>>> G * 4
‘GompeiGompeiGompeiGompei’
>>> G
‘Gompei’
```

There are two very important concepts present in the eight lines above.

Any ideas?

Strings: Polymorphism

The first important concept is that the plus sign (+) can mean different things for different objects. It means addition for numbers, but concatenation for strings.

We call this property ***polymorphism***, and it is a general property in Python. The ideas behind this are beyond the scope of our introduction to Python today, but you should be aware of it.

Strings: Immutability

The second concept from the string slicing is ***immutability***.

```
>>> G * 4
'GompeiGompeiGompeiGompei'
>>> G
'Gompei'
```

Even after the operations on `G`, the original string object remained the same.

Every string operation in Python is designed to create a new string object, because strings in Python are ***immutable***.

This means that once a string object has been created, it cannot be changed in place.

Try replacing one of the positions in a string sequence. What happens?

Strings: Immutability

Every object in Python is either immutable, or not.

Immutable objects are: numbers, strings, tuples

Mutable objects are: lists, dictionaries, sets

This consistency of certain objects can be used to guarantee that an object remains constant throughout your program.

Mutable objects could change at any time, by accident or intentionally, creating unexpected results if you aren't careful.

Again, the details are beyond the scope of this introduction, but ***immutability*** is an important concept you need to be aware of with Python.

String Specific Methods

String manipulation in Python can be very powerful, even just using the built-in methods and nothing else.

Let's run through some examples of string specific methods:

```
>>> G = 'Gompei'
>>> G.find('mp')
1
>>> G
'Gompei'
>>> G.replace('ei', 'Bot')
'GompBot'
>>> G
'Gompei'
```

Again, despite what you may think, we did not change the original string `G`. If we wanted to keep the results of our `.replace()` operation, we would need to create a new string object to hold it.

String Specific Methods


```
>>> line = ‘Gompei for President 2016’
>>> line.split(‘ ’)
[‘Gompei’, ‘for’, ‘President’, ‘2016’]
>>> G = ‘Gompei’
>>> G.upper()
‘GOMPEI’
>>> G.isalpha()
True
>>> line = ‘Vote Goat 2016!\n’   #The \n is a newline
>>> print line
Vote Goat 2016

>>> line.rstrip()  #Removes whitespace characters on the right side
‘Vote Goat 2016!’
>>> line.rstrip().split(‘ ‘)
[‘Vote’, ‘Goat’, ‘2016’]
```

Lists

Lists in Python are also generally sequences, just as strings are.

Lists are *positionally ordered* collections of *arbitrarily typed* objects.

Lists do not have a fixed size (remember, they are *mutable*).

Because of their mutability, we can modify lists in place, providing a very flexible tool for representing arbitrary collections.

Since lists are sequences just like strings, lists support all of the sequence operations we discussed for strings.

Lists


```
>>> L = [3.1415, ‘gomp’, 42]
>>> len(L)
3
>>> L[0]
3.1415
>>> L[:-1]
[3.1415, 42]
>>> L + [‘life’, ‘universe’, ‘everything’]
[3.1415, ‘gomp’, 42, ‘life’, ‘universe’, ‘everything’]
>>> L
[3.1415, ‘gomp’, 42]
```

List Operations

Lists look a bit like arrays, but they are more flexible.

Arrays in other programming languages (typically) have fixed types for all array elements.

Python arrays can contain pretty much any sort of mixture of objects you want.

The ability to resize arrays in Python is especially useful, since most other languages require allocating arrays to a fixed size ahead of time.

```
>>> L.append('GompBot')
[3.1415, 'gomp', 42, 'GompBot']
>>> L.pop(2)
42
>>> L
[3.1415, 'gomp', 'GompBot']
```

List Operations


```
>>> Q = [‘zz’, ‘bb’, ‘aa’, ‘qq’]
>>> Q.sort()
>>> Q
[‘aa’, ‘bb’, ‘qq’, ‘zz’]
>>> Q.reverse()
>>> Q
[‘zz’, ‘qq’, ‘bb’, ‘aa’]
```

The `sort()` method orders the list in ascending order, while `reverse()` does the opposite.

In each case, the method modifies the list directly

List Nesting

Lists in Python can also be nested, which can be used to represent matrices if you want.

In practice, you would use available packages like Numpy to do matrix operations, but still, you could do it if you wanted.

Either way, a nested list for any purpose can be indexed like this:

```
>>> M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1]
[4, 5, 6]
>>> M[1][2]
6
```

Lists: Comprehensions

One of the more useful features in Python that we haven't touched on yet is *iteration*.

Sequences in Python are ***iterable***, meaning there is a sort of built-in mechanism for looping or iterating on lists.

A first example of this is ***list comprehensions***

```
>>> col2 = [row[1] for row in M]
>>> col2
[2, 5, 8]
```

So what you can see is that ***list comprehensions*** build new ***lists***.

Lists: Comprehensions

We can do more complicated things in list comprehensions though, like modify the new list as it is being built:

```
>>> colX2= [row[1] * 2 for row in M]
>>> colX2
[4, 10, 16]
```

Or we can filter out results we don't want:

```
>>> [row[1] for row in M if row[1] % 2 == 0]
[2, 8]
```

Things can get out of hand quickly:

```
>>> [[x, x/2, x*2] for x in range(-6, 7, 2) if x > 0]
[[2,1,4],[4,2,8],[6,3,12]]
```

Lists

We can also just loop over lists using `for`:

```
#!/usr/bin/env python
L = [1, 2, 3, 4, 5, 6]
sum = 0
for num in L:
   sum += num
print sum
```

You can see the same iterative pattern over the list elements as we did in the list comprehensions.

Files

File objects in Python are the main interface to external files on your computer. We can read and write to a huge variety of files, including text files, Excel spreadsheets, audio files, emails, etc, etc.

To create a file you call the built-in `open` function, passing in an external filename and an option 'mode' such as read or write.

```
>>> f = open('data.txt', 'w')
>>> f.write('Hello\n')
>>> f.write('world\n')
>>> f.close()
```

These commands create a file in your current directory and writes out the text to that file.

To read a file back in, open it with a `'r'` processing mode. The read mode is the default if you do not specify one.

Files

To read a file back in, open it with a `'r'` processing mode. The read mode is the default if you do not specify one.

```
>>> f = open('data.txt')
>>> text = f.read()
>>> text
>>> 'Hello\nworld\n'
>>> print(text)
Hello
world
```

Files

To read a file back in, open it with a `'r'` processing mode. The read mode is the default if you do not specify one.

```
>>> f = open('data.txt')
>>> text = f.read()
>>> text
>>> 'Hello\nworld\n'
>>> print(text)
Hello
world
>>> text.split()
['Hello', 'world']
```

Functions

A function is a device that groups a set of statements together so that they can be run as part of a more complicated program.

Functions are the most basic program structure in Python, and provide the best way to reuse code that you write.

Statement or expression	Example Literals/Creation
Call expressions	`myfunc('gompei', 42)`
def	`def printer(message)` `  print('Hello' + message)`
return	`def adder(a, b=1)` `  return a + b`

Functions


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

name = raw_input ("Input your name : ")
def namePrinter(name):
   greeting = "Hi, " + name
   return greeting

namePrinter(name)
```

Practice!

The only way to get better at programming is to write code!

Now that you know the basics, you need to practice writing some code, trying things out, solve some problems.

Here are some choices for how you will spend the next hour:

1. Maths!
2. Games!
3. Sleep!
4. Something else!

Practice!

Maths:

1. Write a program that, given a list of numbers, returns a list where all adjacent == elements have been reduced to a single element (e.g. [1, 2, 2, 3] returns [1, 2, 3])
2. Write a program that computes the factorial of a number
3. Write a program that calculates the sum of Fibonacci numbers below a threshold
4. Write a program that takes as input from the user the radius of a circle, and outputs the area of the circle.

Games:

1. Write a guessing game program, where the computer holds a random number between 1 and 100, and you get a certain number of tries to guess the number
2. Write a program that plays a simple version of hangman
3. Write a program that plays Battleship

Getting Started: Interactive Prompt

On the workstations, open the Anaconda Prompt

This is the same as the standard Windows command prompt, but includes hooks into the Anaconda Python installation.

Once you have the command prompt open, simply type the command `python`

The command prompt will change from > to >>> after printing out some version information

This is the **interactive** Python shell

Here, you can set variables, print the variables, do basic arithmetic, etc.

Setting variables in Python is as easy as using an equal sign (=)

Printing variables in Python uses the `print` command

Getting Started: Python Scripts

On the workstations, open Notepad++ (not Notepad)

From here, we are going to recreate out last example, but instead save the commands inside of a Python script

Just as with bash or csh shell scripts, we include a line to tell the OS what kind of interpreter to run

At the top of your file, type:

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

Now put in your commands from the interactive prompt, such as setting the values of `mascot`, `a`, `b`, `f`, etc, as well as all the `print` statements (we actually need them now!)

Save the file to the desktop as `gompei.py`, and switch back to the Anaconda Prompt

Type `exit()` to leave the Python interpreter

Getting Started: Python Scripts

From Anaconda Prompt, type:

```
python gompei.py
```

Instead of opening up a Python command prompt, the Python interpreter will now run through your file, executing the commands one at a time.

These two ways of using Python, interactive commands and file based programs, allow you to quickly test and debug commands, and then save them to a file as part of a program.

Getting Started: Module Basics

Up to this point we have just typed single commands for Python to execute for us.

We have seen how we can combine commands in a file to build up programs.

What if we had some really useful function we wrote, or some static data saved in a dictionary that we wanted to use in our program?

One of the most important aspects of Python is the idea of encapsulation, and more specifically the **`import`** of modules, which builds up the Python *program architecture*

We haven't imported any modules yet, but you have already made one!

Every Python source code file that ends with a *.py* extension is a module.

No special code or syntax is required, and any other file can access the items in a module by *importing* that module

Getting Started: Module Basics

So what happens when a module is imported into another Python program?

All of the code in the module is run, from start to finish.

This makes all of the variables and functions available in the program doing the `import`

Try this out in an interactive session.

Open a Python command prompt, and type `import gompei`

What happens?

Getting Started: Module Basics

Ok, now all of our variables and functions are supposed to be available.

Try printing `mascot`. What happens?

Getting Started: Module Basics

All of the variables are there, just not how you might expect.

Modules are meant to be a library of tools, *general* routines or functionality that you should never have to code again!

Modules are just a package of variable names, what is known as a *namespace*

The names contained in this package are the *attributes* of that namespace

An *attribute* is a *variable* attached to a specific *object*

So where did our variables like `mascot` go, and how do we access them?

We need to access the *attributes* of the module *object*. In Python, *everything is an object*

The steps are `import`, then *qualify* the module name with the *attribute* you want:

```
>>>import gompei
Gompei
…
8
>>>gompei.mascot
Gompei
```

Getting Started: Module Basics

We can skip the whole qualifying the object business if we know what we want directly from the module.

To get something specific from a module, we can use the `from` statement

```
>>>from gompei import mascot
>>>print mascot
Gompei
```

All this really does is make a copy of that specific *attribute* in our current program *namespace,* creating a new *variable (object)*

Getting Started: Module Basics

To look inside a module and see what it contains, try the `dir()` function:

```
>>> import gompei
>>> dir(gompei)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'a', 'b', 'c', 'f', ‘mascot']
```

We can see all the variables we defined, but a bunch of other stuff as well.

All of the other stuff (with the underscores) are *built-in* names that are always predefined by Python, and each has a special meaning (not covered today).

Summary: Modules are a package of *variables*, contained in an *independent namespace*

Modules are the largest program structure in Python