Lab 1

Getting Started

Lab Objective: *Introduce basic coding procedures and objects usage in Python.*

Python

Python is a powerful general-purpose programming language. As an interpreted language, it can be used interactively. It is quickly gaining momentum as a fundamental tool in scientific computing because it has the following features:

- Clear, readable syntax
- Full object orientation
- Complete memory management (via garbage collection)
- High level, dynamic datatypes
- Extensibility via C
- Ability to interface with other languages such as R, C, C++, and Fortran
- Embeddability in applications
- Portability across many platforms (Linux, Windows, Mac OSX)
- Open source

In addition to these, Python is freely available and can also be freely distributed.

Running Python

Python 2.7 is required for the labs in this text and can be downloaded from `http://www.python.org/`. Although later versions of Python are available, they do not have many of the features needed for scientific computing.

Many free IDEs (Integrated Development Environments) and text editors are compatible with Python. We recommend you use IPython, which provides three

different interfaces: commandline, QTConsole, and Notebook. You can open these interfaces by running `ipython`, `ipython qtconsole`, or `ipython notebook` respectively. The commandline interface is the simplest of the three, as it merely adds colored syntax to the text in the terminal window. The QTConsole provides some extra features not available in the commandline interface. The Notebook interface has the most features and is displayed in a web browser.

For more information on installing Python and various libraries, see Appendices A and B.

Learning Python

The remainder of this lab will introduce you to some basic Python data types and control flow blocks. The text for this lab is intended to tell you just enough so that you can create these objects and experiment with them. Each section has additional required readings, listed at its end. Moreover, as you begin your study of Python, we *strongly* suggest you read the following:

1. Chapters 3, 4, and 5 of the Official Python Tutorial (`http://docs.python.org/2.7/tutorial/introduction.html`).

2. Section 1.2 of the SciPy Lecture Notes (`http://scipy-lectures.github.io/`).

3. PEP8 - Python Style Guide (`http://www.python.org/dev/peps/pep-0008/`).

In addition to these resources, there are other ways to learn Python. One useful aspect of the IPython interfaces is *object introspection*, which allows you to see all the methods associated to an object. Also, you can use a question mark to learn about an object or method.

```
# You can use a pound sign to write a single-line comment.

# To see the methods associated to an object, type the object name followed by a ↩
    period, and press tab.
>>> list.
list.append   list.extend   list.insert   list.pop      list.reverse
list.count    list.index    list.mro      list.remove   list.sort

# To learn about a method, use "?".
>>> list.append?
Type:       method_descriptor
Base Class: <type 'method_descriptor'>
String Form:<method 'append' of 'list' objects>
Namespace:  Python builtin
Docstring:  L.append(object) -- append object to end
```

Finally, if you cannot answer your question using these strategies, try googling it. Many common questions about Python programming have been answered on internet forums.

Data Types

Numerical Types

There are four numerical data types, `int`, `long`, `float`, and `complex`, each of which stores a certain kind of number.

```
>>> type(3)
int

>>> type(3.0)
float
```

Python can be used as a calculator. Use `**` for exponentiation.

```
>>> 3**2 + 2*5
19
```

We can also create and manipulate variables with Python.

```
# Use a SINGLE equals sign to create a variable.
>>> x = 12
>>> y = 2 * 6

# Use a DOUBLE equals sign to check equality of variables.
>>> x == y
True
```

Required readings:

- `http://scipy-lectures.github.io/intro/language/basic_types.html#numerical-types`
- `https://docs.python.org/2.7/tutorial/introduction.html#numbers`
- `https://docs.python.org/2.7/library/stdtypes.html#typesnumeric`

Problem 1. Use the required readings and the strategies listed in the section "Learning Python" to help you answer the questions in this lab.

1. How do you convert an integer to a float? (This is called *casting*.)
2. What are the two ways to create a complex number? How do you extract the real part or the imaginary part?
3. The way the operator `/` behaves depends on the types of its operands.
   (a) Why does `7/3` return 2?
   (b) How can you get decimal answers to a division problem like 7/3?
   (c) How can you get integer answers when you divide floats?

Strings

A Python `string` can be created with either single or double quotes. They can be concatenated with the `+` operator.

```
>>> str1 = "I love"
>>> str2 = 'the ACME program'
>>> my_string = str1 + " " + str2 + "!"
>>> mystring
'I love the ACME program!'
```

We can access single characters of strings using brackets and a range of characters using *slicing*. Slicing syntax is `[start:stop:step]`. The parameters `start` and `stop` default to the beginning and end of the string, respectively. The parameter `step` defaults to 0. For more information on slicing, see the section "Lists."

```
# Indexing begins at 0 and negative numbers count backwards from the end.
>>> my_string[-1]
'!'

# Pick out every other character in a string.
>>> my_string[::2]  # string[start:stop:step]
'Ilv h CEporm'
```

Required readings:

- `http://scipy-lectures.github.io/intro/language/basic_types.html#strings`
- `https://docs.python.org/2.7/tutorial/introduction.html#strings`

Problem 2. Answer the following questions about strings.

1. Suppose `my_string = "I love the new ACME program!"`. What output is produced by `my_string[::3]` and `my_string[::-1]`?
2. How can you print a `string` backwards?

Lists

A Python `list` is created by enclosing comma-separated values with square brackets. You can access a single entry of a `list` or a range of entries with the same indexing or slicing operations as we used on strings.

```
>>> my_list = ["Remi", 21, "08/06", 1993]
>>> my_list
['Remi', 21, '08/06', 1993]
>>> my_list[-2]
'08/06'
```

Whenever possible, you should create your lists using a *list comprehension*, which is demonstrated below.

```
# The command range(n) produces the list [0, 1, 2, . . . , n-1].
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Required readings:

- `http://scipy-lectures.github.io/intro/language/basic_types.html#lists`
- `https://docs.python.org/2.7/tutorial/introduction.html#lists`
- `https://docs.python.org/2.7/tutorial/datastructures.html#list-comprehensions`

Problem 3. Answer the following questions about lists.

1. If `my_list = ["mushrooms", "rock climbing", 1947, 1954, "yoga"]`, write a sequence of commands that does the following:
   (a) Finds the length of the list,
   (b) Appends "Jonathan, my pet fish" to the end of the list,
   (c) Inserts `'pizza'` at index 3, and
   (d) Clears the entire list.
2. Write a sequence of commands that does the following:
   (a) Creates an empty list called `num`,
   (b) Adds the integers `3`, `5`, `19`, `20`, and `4` to `num`,
   (c) Replaces the integer at index 3 with itself converted to a string,
   (d) Removes the integer at index 2, and
   (e) Sorts `num` backwards.
3. Using a list comprehension, create a list that has the elements of `num` converted into strings. For example, if `num = [ 5, 4, 3, 2, 1]`, write a list comprehension that outputs `num=['5', '4', '3', '2', '1']`.

Sets

A Python `set` is an unordered collection of distinct objects, and can be created from a list. We can add or remove members from a set after its creation.

```
>>> gym_members = set(["Doe, John", "Doe, John", "Smith, Jane", "Brown, Bob"])
>>> gym_members
set(['Brown, Bob', 'Doe, John', 'Smith, Jane'])
>>> gym_members.discard("Doe, John");
```

```
>>> gym_members
set(['Brown, Bob', 'Smith, Jane'])
>>> gym_members.add("Lytle, Josh")
>>> gym_members
set(['Brown, Bob', 'Lytle, Josh', 'Smith, Jane'])
```

Like mathematical sets, a `set` has operations like union, intersection, difference, and symmetric difference.

```
# Set intersection returns a new set object.
>>> library_members = set(["Lytle, Josh", "Henriksen, Ian", "Grout, Ryan"])
>>> set.intersection(gym_members, library_members)
set(['Lytle, Josh'])
```

Required readings:

- `http://scipy-lectures.github.io/intro/language/basic_types.html#more_container_types`
- `https://docs.python.org/2.7/tutorial/datastructures.html#sets`

Problem 4. Answer the following questions about sets.

1. What are two ways to create sets? How do you create an empty set?
2. Define two sets and use a Python command to find their union.

Dictionaries

Like a `set`, a Python `dictionary` is an unordered data type. A `dictionary` stores `key :value` pairs, which are called *items*. The values of a dictionary are indexed by its keys.

```
>>> tel = {"marriott": 4121, "math": 2061, "visual arts" : 7321}
>>> tel["math"]
2061
```

The keys of a dictionary must be *immutable*, which means that they must be objects that cannot be modified after creation. Numerical types and strings are immutable objects. Lists, dictionaries, and sets are mutable.

Required readings:

- `http://scipy-lectures.github.io/intro/language/basic_types.html#dictionaries`
- `https://docs.python.org/2.7/tutorial/datastructures.html#dictionaries`

Problem 5. Answer the following questions about dictionaries.

1. What are two ways to create dictionaries? How do you create an empty dictionary?
2. How do you delete a key-value pair from a dictionary?
3. How do you access a list of all the values in your dictionary? How do you access a list of all the items?

Control Flow Tools

Control flow blocks control the order in which your code is executed. Python supports the usual control flow statements including while loops, if statements, for loops, and function definitions. For examples besides those in this lab, see the following resources:

- Sections 1.2.3.1-1.2.3.4 of `http://scipy-lectures.github.io/intro/language/control_flow.html`
- `http://scipy-lectures.github.io/intro/language/functions.html`
- `https://docs.python.org/2.7/tutorial/introduction.html#first-steps-towards-program`
- Sections 4.1-4.3, 4.6, and 4.7.1-4.7.2 of `https://docs.python.org/2.7/tutorial/controlflow.html`

The While Loop

Python uses indentation to identify the beginning and end of blocks of code, so you must indent each line of an execution block the same way. The convention is to indent blocks of code with four spaces. A `while` loop executes an indented block of code *while* the given condition holds.

```
>>> i = 0
>>> while i < 10:           # The colon is required.
...     print i,             # This indented line is executed if b<10.
...     i = i+1       # This indented line is also executed if b<10.
...
0 1 2 3 4 5 6 7 8 9
```

In the above example, the comma in the line `print i,` makes Python print all the numbers on the same line (by stripping off newline characters). Try running this example without the comma and see what happens.

The If Statement

An `if` statement executes the indented code *if* the given condition holds. The `elif` statement is short for "else if" and can be used multiple times following an `if`

statement, or not at all. The `else` keyword may be used at most once at the end of a series of `if`/`elif` statements.

```
>>> food = "bagel"         # Use a SINGLE equals sign create variables
>>> if food == "apple":    # Use a DOUBLE equals sign to check equality
...     print "72 calories"
... elif food == "banana":
...     print "105 calories"
... elif food == "egg":
...     print "102 calories"
... else:
...     print "calorie count unavailable"
...
calorie count unavailable
```

The For Loop

A `for` loop iterates over the items in any *iterable*. Iterables include lists, sets, and dictionaries.

```
>>> for i in range(10):
...     print i,
...
0 1 2 3 4 5 6 7 8 9
```

Function Definition

To define a function, use the `def` keyword followed by the function name and a parenthesized list of formal parameters. Then indent the function body.

```
# Compute the area of a rectangle
>>> def area(width, height):
...     return width*height
...
>>> area(2, 5)
10
```

We define a function with *parameters* and call it with *arguments*. In the example above, `width` and `height` are parameters for the function `area`. The values `2` and `5` are the arguments that we pass when calling the function. In practice, the terms *parameter* and *argument* are often used interchangeably.

It is also possible to specify default values for formal parameters, as in the following example.

```
>>> def fn(a, b, c=0):
...    print a, b, c
```

The function `fn` has three formal parameters, and the value of `c` defaults to 0. We can pass arguments to `fn` based on position (positional arguments) or name (named arguments or keyword arguments). We must define positional arguments before keyword arguments.

```
# Call fn with 2 positional arguments (c=0 by default)
>>> fn(1, 2)
1 2 0

# Call fn with 3 positional arguments
>>> fn(4, 5, 6)
4 5 6

# Call fn with 1 positional argument and 2 named arguments
>>> fn(1, c=2, b=3)
1 3 2
```

The final example demonstrates the flexibility of Python but is somewhat confusing. Whenever possible, you should pass arguments to a function in the order they are defined in the function. Thus, do the following.

```
# Call fn with 1 positional argument and 2 named arguments
>>> fn(1, b=3, c=2)
1 3 2
```

Problem 6. Define the following function in your interpreter.

```
>>> def track(n, my_list=[]):
...     my_list.append(n)
...     return my_list
```

1. Now execute the following commands in your interpreter: `track(1)`, `track(2)`, `track(3)` and `track(1, [1, 2, 3])`. What outputs do you get?
2. The default values of a function are only evaluated once. This means that when you call `track` with the default value for `my_list`, the same list is used each time. Modify `track` so that the default value for `my_list` is not shared between calls. With your modified function, the outputs in part (a) should be `[1]`, `[2]`, `[3]` and `[1, 2, 3, 1]`. Hint: Read here about the Python constant `None`: `https://docs.python.org/2/library/constants.html`.

The most general form of a function definition is as follows.

```
def fn(*args, **kwargs):
```

This means that "`fn` takes some arguments and keyword arguments." The arguments, `args`, are stored as a tuple; and the keyword arguments, `kwargs`, are stored in a dictionary. The function `fn` can accept any number of arguments and keyword arguments.

```
>>> def fn(*args, **kwargs):
...     print "Positional: ", args
...     print "Keyword: ", kwargs
```

```
...
>>> fn("Hello", 2, 1, apples = 3, oranges = 2)
Positional: ('Hello', 2, 1)
Keyword: {'apples': 3, 'oranges': 2}
```

Problem 7. 1. Explain what the `print` and `return` statements do. How are they different?

2. Modify the code below so that it produces the desired output.

```
Grocery List = ['pineapple', 'orange juice', "avocados", "pesto ↩
    sauce"]
for i in range(Grocery List)
if i % 2 = 0
print i, Grocery List(i)
```

Desired output:

```
0 pineapple
2 avocados
```

3. When you call the function "groceries" below, it returns an error. Why?

```
>>> def groceries(food, drink):
...     print food
...     print drink

>>> groceries(food="Bananas", "Juice")
```