Lab 1

Python: Getting Started

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

Prerequisites

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

There are many IDEs (Integrated Development Environments) and text editors freely available. We recommend you use the IPython. IPython provides three different interfaces: commandline, QTConsole, and Notebook. The commandline interface is the simplest of the three. It displays code in nice colorized terminal and it very easy to use. It can be started by simply running `ipython`. The QTConsole is an enriched commandline interface. It provides some extra features not available in the commandline interface while be just as easy to use. You can run the QTConsole interface by executing `ipython qtconsole`. The Notebook interface provides the most comprehensive list of features and is display via a web browser. The Notebook interface can be launched by running `ipython notebook`. This will start the Notebook server and launch a web browser with the interface.

For more information on downloading and installing various packages, please reference the Appendix.

Python

Python is a powerful general-purpose programming language. It is an interpreted language and can even be used interactively. It is quickly gaining momentum as a fundamental tool in scientific computing and has several very nice key features:

- Clear, readable syntax
- Full object orientation

- Complete memory management (via garbage collection)
- High level, dynamic datatypes
- Extensibility via C
- Ability to interface to 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.

Learning Python

The following examples and problems highlight important functionality and syntax in Python, but they are by no means comprehensive. As you familiarize yourself with Python and begin to work through the problems, 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/`).

Python is a high-level programming language with powerful capabilities. It has many standard data types including numbers, strings, lists, sets, and dictionaries.

Numbers

Example 1.1. As its name suggests, number data types store numeric values. They are immutable data types, meaning that changing the value results in a newly allocated variable. There are four different numerical types: `int`, `long`, `float`, and `complex`

```
# Comments will frequently be provided in this form.
>>> '''
... It is also possible to use multiline strings as comments.
... These are enclosed by 3 single (or double) quotations at
... the beginning and end.
... '''

# Python can be used as a simple calculator.
>>> 9*8 + 4*5
92

# Integer division returns the floor.
```

```
>>> 100/3
33

>>> # Use floating points to ensure the floating point operand.
>>> 100/3.0
33.333333333333336

# Use variables to store values.
>>> x = 12
>>> y = 2 * 6

# It is possible to check equality between two variables.
>>> x == y
True

# It is also possible to assign multiple values to multiple variables
>>> a, b, c = 1, 2, 3
>>> (a + b)**c       # Exponentiation is denoted '**'
27
```

Problem 1. It is very useful to know how to find help in the official Python documentation (`https://docs.python.org/2.7/`). Use the official Python documentation to find the answer to each of the following:

1. What are the two ways to create a complex number? How do you extract just the real part and just the imaginary part?
2. How would you cast an integer as a float?
3. Is there a way to explicitly express integer division when using floats?

Problem 2. Why does `7/3` return 2 in Python 2.7?

Strings

Example 1.2. A string in Python is an ordered sequence of characters between quotation marks. Those quotation marks may be single or double quotes, but must be the same at both ends of the string. Strings can be concatenated using the `+` operator. We can also access substrings and individual characters using brackets. To do this we identify the n^{th} character of string, S, by `S[n-1]`, or the i^{th} through the j^{th} character by `S[i:j+1]`. The index of a string of length n starts with zero and goes to $n-1$ at the end. Python allows the use of negative indices to index the from the end of a string. The last character of any string can be accessed by $n-1$ or -1. All ranges in Python include the origin point and exclude the end point Also, the string datatype is immutable, meaning it cannot be modified in place once it has been allocated. Changing a string creates a new copy of the string with the change. We can cast other values in Python, like integers and to strings using the `str()` function.

```
>>> str1 = "I love"
>>> str2 = "the ACME program"
>>> str3 = "puppies and baby ducks"
>>> str4 = "sleeping"

# The + operator concatenates strings.
>>> str1 + " " + str2 + "!"
'I love the ACME program!'

# To slice, [start:stop:step].
>>> my_string = "T" + str2[1:4] + str3[:9] + "re " + str4 + "."
>>> my_string
'The puppies are sleeping.'
>>> my_string[:-1:2]
'Teppisaesepn'
```

Problem 3. Answer the following questions. Use the official Python documentation, if necessary.

1. What does it mean for a string to be an immutable object?
2. What happens in `my_string[::3]` and `my_string[::-1]` provided that `my_string = "I love the new ACME program!"`?
3. How can you access the entire string in reverse?

Lists

Example 1.3. The list data type is very versatile. It is represented as comma separated values enclosed within square brackets. It is a mutable sequence allowing in-place modification after creation. The values within a list can be different data types and are accessed using the slice operator (just like strings). There are many methods available to the list data type. There are various methods for interacting with lists and some are demonstrated below, including `append`, `insert`, `remove`, `sort`, and `reverse`.

```
>>> my_list = ["Remi", 21, "08/06", 1993]
>>> my_list[0]
'Remi'
>>> my_list[2] =  "07/10"
>>> my_list
['Remi', 21, '07/10', 1993]

# Append adds an item to the end of your list.
>>> my_list.append("student")
>>> my_list
```

```
['Remi', 21, '07/10', 1993, 'student']

# The insert method takes an item and inserts it at a given position.
>>> my_list.insert(0, "female")
>>> my_list
['female', 'Remi', 21, '07/10', 1993, 'student']

# The remove method takes an item and removes it from your list.
>>> my_list.remove("student")
>>> my_list
['female', 'Remi', 21, '07/10', 1993]

# Lists can also be sorted in place. This changes the original list.
>>> my_list.sort()
>>> my_list
[21, 1993, '07/10', 'Remi', 'female']

# Lists can also be reversed.
>>> my_list.reverse()

# List comprehensions provide a powerful way to create lists.
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The last example, list comprehensions, is very important. Whenever possible, you should try to create your lists using list comprehension, rather than with a for loop (for loops will be discussed later in this lab).

Problem 4. Answer the following questions. Use the official Python documentation, if necessary.

1. What is the difference between mutable and immutable objects?
2. If `my_list = ["mushrooms", "rock climbing", 1947, 1954, "yoga"]`
   (a) How would you access "yoga"?
   (b) How would you view a copy of `my_list`?
   (c) How would you clear the entire list?
   (d) How would you find the length?
   (e) How would you assign "mushrooms" (first entry) and "rock climbing" (second entry) of `my_list` to "Peter Pan" and "camelbak"? (This should be done in one line of code and is known as a slice assignment)
   (f) How would you add "Jonathan, my pet fish" to the end of the list?
3. What sequence of commands would you use to implement the following?
   (a) Create an empty list.

(b) Add 5 integers to your list.

(c) Cast the integer at index 3 as a float.

(d) Remove the integer at index 2.

(e) Sort your list backwards.

Sets

Example 1.4. Sets are an unordered collection of distinct objects. It is a mutable object that allows objects to be added and removed after creation. Like mathematical sets, Python supports operations like union, intersection, difference, and symmetric difference.

```
>>> gym_members = set(["Doe, John", "Smith, Jane", "Brown, Bob", "Jones, Sally"])
>>> gym_members
{'Brown, Bob', 'Doe, John', 'Jones, Sally', 'Smith, Jane'}
>>> gym_members.add("Lytle, Josh")
>>> gym_members.add("Doe, John")
>>> gym_members
{'Brown, Bob', 'Doe, John', 'Jones, Sally', 'Lytle, Josh', 'Smith, Jane'}
>>> library_members = set(["Lytle, Jane", "Henriksen, Ian", "Smith, Jane", "Grout↩
    , Ryan"])
>>> library_members.discard("Smith, Jane")
>>> library_members.add("Lytle, Josh")
>>> library_members
{'Grout, Ryan', 'Henriksen, Ian', 'Lytle, Jane', 'Lytle, Josh'}

# Note that, by default, set intersection returns a new set object.
>>> inter = set.intersection(gym_members, library_members)
>>> inter
{'Lytle, Josh'}
>>> library_members & gym_members
{'Lytle, Josh'}

# We can also do set comprehensions like we did list comprehensions.
>>> my_set = set(x**2 for x in range(10))
>>> my_set
{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Problem 5. Answer the following questions. Use the official Python documentation, if necessary.

1. What are the two ways to create sets? How do you create an empty set?
2. How would you take the union of sets?
3. How are sets indexed?

Dictionaries

Example 1.5. Dictionaries are another unordered data type. They are useful for storing key-value pairs.

```
>>> tel = {"accounting":4234, "admissions": 2507, "financial aid": 4104, "↩
    marriott": 4121, "math": 2061, "visual arts" : 7321}
>>> tel["math"]
2061
>>> tel.keys()
['visual arts',
 'admissions',
 'financial aid',
 'accounting',
 'marriott',
 'math']
```

Problem 6. Answer the following questions. Use the official Python documentation, if necessary.

1. What are the two ways to create dictionaries? How do you create an empty dictionary?
2. In Example 1.3 above we created a list using list comprehension. Create a dictionary using a dictionary comprehension to produce the following output:

   ```
   {2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
   ```

   (Note that because dictionaries are unordered, your output may not exactly the output displayed above)
3. How do you delete a key-value pair?
4. 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 used in other languages including while loops, if statements, for loops, and function definitions.

The while Loop

Example 1.6. A while loop executes the indented block of code following it *while* the given condition holds. Python uses indentation to identify the beginning and end of blocks of code, so be careful to indent the same amount of space for each line of an execution block. Most Python code uses a standard of four spaces to indent blocks of code. Be aware that you can create an infinite loop if the condition is always true.

```
# The Fibonacci sequence can be formulated using a while statement.
>>> a, b = 0, 1
>>> while b < 10:           # The trailing colon is required.
...     print a             # This indented line is executed if b<10.
...     a, b = b, a+b       # This indented line is also executed if b<10.
...
0
1
1
2
3
5
```

The if Statement

Example 1.7. An if statement behaves as expected: if the condition holds, execute the following indented code. Note again that the indentation is important to Python. The elif statement is short for "else if" and can be used multiple times or not at all. The else keyword may only be used once at the end (requires no condition), but may also be omitted.

```
>>> food = "bagel"
>>> if food == "apple":
...     print "72 calories"
... elif food == "banana":
...     print "105 calories"
... elif food == "egg":
...     print "102 calories"
... elif food == "oatmeal":
...     print "147 calories"
... elif food == "pizza":
...     print "298 calories"
... else:
...     print "calorie count unavailable"
...
calorie count unavailable
```

The for Loop

Example 1.8. A for loop iterates over the items in any list, set, dictionary, tuple or other iterable. The `range` function is typically used to generate lists that allow iteration over a sequence of numbers. Note that Python stops just *before* the given stop value.

```
>>> for i in range(5):
...     print i
...
0
1
```

```
2
3
4
>>> my_list = ["Henry XXI", "beta fish", "asparagus"]
>>> for x in my_list:
...     print "King" + " " + x
...
King Henry XXI
King beta fish
King asparagus
>>> for i in range(len(my_list)):
...     my_list.append(i)
...
>>> my_list
['Henry XXI', 'beta fish', 'asparagus', 0, 1, 2]
```

Function Definition

Example 1.9. The `def` keyword allows for a function definition. The keyword is followed by the function name and a parenthesized list of formal parameters. Once again, indentation is important.

```
>>> def fibonacci(n):
...     a, b = 0, 1
...     while a < n:
...         print a, b
...         a, b = b, a+b
...
>>> fibonacci(10)
0 1
1 1
1 2
2 3
3 5
5 8
8 13
```

Let's look at another function.

```
def fn(a, b, c=0):
    print "a: {}, b: {}, c: {}".format(a, b, c)
```

The function, fn, has three parameters: a, b, and c. When we call this function, we need to provide arguments for each of these parameters. Note, however, that we have defined a default value for c. Therefore, if we only provide arguments for a and b, Python will use the default value for c (which is 0). We can pass arguments based on position or by name. When called with just a list of values, Python will assign the first argument to the first parameter, the second argument to the second parameter, etc. This is what is meant by positional arguments. We can also explicitly assign arguments by using the parameter's name. These are named arguments. They are also often called keyword arguments. There are a variety of ways to call our function.

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

```
>>> fn(1, 2)
a: 1, b: 2, c: 0

# Call fn with 3 positional arguments
>>> fn(4, 5, 6)
a: 4, b: 5, c: 6

# Call fn with 1 positional argument and 1 named argument (c=0 by default)
>>> fn(15, b=16)
a: 15, b: 16, c: 0

# Call fn with 1 positional argument and 2 named arguments
>>> fn(12, c=13, b=14)
a: 12, b: 14, c: 13

# Call fn with 2 named arguments (c=0 by default)
>>> fn(b=10, a=11)
a: 11, b: 10, c: 0

# Call fn with 3 named arguments
>>> fn(b=7, a=8, c=9)
a: 8, b: 7, c: 9
```

Notice that in every case, we must define any positional arguments first. The way we call functions in Python can be very flexible. However, to avoid confusion, it is common practice to give arguments to a function in the order they are defined in the function definition whenever possible.

Note

When defining a function, we define the *parameters* of the function. We then call the function with *arguments*. While the distinction often isn't very important, in some cases, it might help you avoid confusion. In practice, though, the two terms are often used interchangeably and are viewed as synonymous.

```
def add_numbers(a, b):
    return a + b
add_numbers(5.2, 10)
```

In the example above, a and b are parameters for the function `add_numbers`. The values 5.2 and 10 are the arguments that we pass when calling the function.

Problem 7. Define the following function in your interpreter.

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

What does the function do? Try calling the function multiples times with different values of n. Print the list returned each time. The behavior of this function demonstrates something very important. The default values of a function are only evaluated once. This means that the same list is being shared for every call of this function. What would be the best way to modify the function so that the list is not shared between calls of the function?

Sometimes when looking at the documentation of a function or when defining your own function you will see the following:

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

This is the most general form of a function definition. Basically, it means that "the function 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 above can accept any number of arguments or keyword arguments.

```
>>> def f(*args, **kwargs):
...     print "Positional: ", args
...     print "Keyword: ", kwargs
...
>>> f("Hello", 2, 1, apples = 3, oranges = 2)
Positional: ('Hello', 2, 1)
Keyword: {'apples': 3, 'oranges': 2}

# The stars are operators and have special meaning in this case
# These operators unpack tuples and dictionaries into arguments or keyword ↩
    arguments
>>> f(*range(5)) # Equivalent to f(0, 1, 2, 3, 4)
Positional: (0, 1, 2, 3, 4)
Keyword: {}
>>> d = {'a': 1, 'b': 2}
>>> f(**d) # Equivalent to f(a=1, b=2)
Positional: ()
Keyword: {'a': 1, 'b': 2}
```

Problem 8. Answer the following questions. Use the official Python documentation, if necessary.

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

2. What is wrong with the following code?

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

   provided you want the following output:

```
0 pineapple
2 avocados
```

Note that this code may contain one error, many errors, or no errors at all.

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")
```

Specifications

We suggest that you submit your solutions in a file called `solutions.py`, using the following format.

```
# Problem 1
1.
2.
3.


# Problem 2
1.


# Problem 3
1.
2.
3.


# Problem 4
1.
2.
3.

# Problem 5
1.
2.
3.

# Problem 6
1.
2.
3.
4.

# Problem 7
1.
```

```

# Problem 8
1.
2.
3.
```

template.txt