A Brief Introduction to Python

Python is an interpreted language, meaning that a program, i.e., the interpreter, is used to execute your commands one after the other. The commands can be entered interactively or they can be stored in a file and the interpreter can act on the file. An interpreted language is distinct from a compiled language such as C, C++, and Java.

With a compiled language (such as C) the commands are first stored in a file, often referred to as the “source code.” The source code is run through a program known as a compiler—this is the act of compiling the code. The result of the compilation is a stand-alone program often referred to as “the executable.” When the executable is run it carries out the operations specified in the source code. One does not interactively execute C, C++, or Java commands (although, arguably, using another program, known as a debugger, one can step through these commands in something like an interactive fashion, but this is very cumbersome).

1 Python as a Calculator

Let us assume we have started an interactive Python session. The interactive prompt is “>>>”. When we see that prompt the Python interpreter is waiting for our input (as we will see later, when we are entering a “block” of instructions or a statement that spans multiple lines, the prompt will change to “...”). We can use Python as a calculator. For example, at the interactive prompt we could type “2 + 3” and then hit the Enter (or return) key. The result will be displayed on the next line:

>>> 2 + 3
5

(Note that the user input will generally be shown in bold.) In Python “+” is used for addition, “-” for subtraction, “*” for multiplication, “/” for division, and “**” for exponentiation. Python can handle integers of arbitrary length:

>>> 3**100
515377520732011331036461129765621272702107522001

We are used to thinking about different types of numbers. Similarly, Python has different ways of storing and representing numbers depending on the number’s “type.” There are integers, floating point numbers (i.e., real numbers), complex numbers, and more. As an example, we can multiply or divide the following two complex numbers1:

>>> (1 + 4j) * (3 - 2j)
(11+10j)

>>> (1 + 4j) / (3 - 2j)
(-0.3846153846153846+1.076923076923077j)

---

1 As you might expect if you’re familiar with imaginary numbers, the imaginary part is associated with the part that has j in it, where j represents the square root of -1 so that j^2 = -1. Thus 3-2j has a real part of 3 and an imaginary part of -2. In Python you cannot write the j before the numeric part, i.e, 3 - j2 is not correct.

---
In some ways the core Python language has rather limited capabilities. However, Python has a huge number of libraries (or modules) we can load to expand its capabilities. There are libraries for handling email, taking apart Web pages, interacting with the underlying operating system, and on and on. The `math` module contains various functions (corresponding to all the functions in the standard C math library). One of the ways to load the `math` module is

```python
>>> import math
```

It seems that nothing has happened, but we have actually introduced quite a number of new functions to Python. We can find out more information by using the built-in `help()` function. Here is part of the information that is displayed\(^2\):

```python
>>> help(math)

NAME
   math

FILE
   /Library/Frameworks/Python.framework/Versions/3.1/lib/
   python3.1/lib-dynload/math.so

MODULE DOCS
   http://docs.python.org/library/math

DESCRIPTION
   This module is always available. It provides access to the
   mathematical functions defined by the C standard.

FUNCTIONS
   acos(...)
      acos(x)

      Return the arc cosine (measured in radians) of x.

   acosh(...)
      acosh(x)

      Return the hyperbolic arc cosine (measured in radians) of x.

   asin(...)
      asin(x)

      Return the arc sine (measured in radians) of x.

   ...
   ...

This description goes on for quite a while. Once it has generated one screen’s worth of output it will pause. Hit the space bar to see more output or hit the Q key to quit. In this description of the

\(^2\)Information given for the “FILE” will vary slightly from system to system.
math module you would eventually see that there is a function called \texttt{sqrt()} that computes the square root of a number. Because of the way we imported this module, we have to put the module name first, a dot, and then the name of the function we want to use\textsuperscript{3}. Thus, to obtain $\sqrt{2}$ we would type

```python
>>> \texttt{math.sqrt(2.0)}
1.4142135623730951
```

What about the square root of a complex number? One of the great features of the interactive Python interpreter is that there is absolutely no harm in trying things. They may work and they may not, but you’re in no danger of crashing your computer or even crashing the interpreter. Let’s try the square root of $3+2j$:

```python
>>> \texttt{math.sqrt(3 + 2j)}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can’t convert complex to float; use abs(z)
```

That clearly didn’t work. Python provided us with an error message but you may find that, at first, any error messages you receive are rather cryptic. Nevertheless, you will certainly know something went wrong. The problem is that the \texttt{math} module was designed for real numbers (or integers)\textsuperscript{4}.

If we want to manipulate complex numbers with various mathematical functions such as square root, we need to use the \texttt{cmath} module (which is specifically designed for complex numbers, hence the leading “c” in the module name):

```python
>>> \texttt{import cmath}
>>> \texttt{cmath.sqrt(3 + 2j)}
(1.8173540210239707+0.55025052270033747j)
>>> \texttt{cmath.sqrt(2.0)}
(1.4142135623730951+0j)
```

From the previous statement you can see that it is okay to use functions from the \texttt{cmath} module on arguments that are real numbers.

## 2 Variables, the Assignment Operator, and the \texttt{print()} Function

Python allows us to define variables that can “point to” any sort of underlying value. In the following we define the variable “a” which points to the integer 2 and the variable “b” which points to the complex number $3.0 + 2.0j$. We then add these values:

```python
>>> a = 2
>>> b = (3.0 + 2.0j)
>>> a + b
(5+2j)
```

\textsuperscript{3}There are other ways to import modules that can change how we access the functions in that module. That will not be discussed in these notes.

\textsuperscript{4}Real numbers in Python are known of \texttt{floats} owing to their “floating” decimal point.
(Note that the parentheses are not required in the assigned of the complex number to the \(b\) variable.) There is an important difference between the equal sign as it is used in Python and how you are probably used to thinking of the equal sign (this distinction is true in most programming languages). In Python the equal sign \textit{does not} mean that the thing on the right is equal to the thing on the left. Instead, you should think of the equal sign as “the assignment operator.” It means that Python should first evaluate the expression on the right, then, after determining what that produces, set the variable on the left equal to result of that expression. For example, in your math classes this makes no sense \(a = a + 1\). However, an expression like that makes perfect sense in Python.

To see the value of a variable, we can either enter it on a line by itself or use a \texttt{print} function to display its value\footnote{In this example there is essentially no different between typing the variable on a line by itself or printing its value. When we get to strings in Sec. 4 we will see that there can be a difference in the behavior of these two ways of displaying a variable.}:

```python
>>> a = 2
>>> a = a + 1
>>> a
3
>>> print(a)
3
```

The value of multiple variables can be displayed with a single \texttt{print} function by separating the list of variables by commas. For example:

```python
>>> a = 5
>>> b = -3.4
>>> c = a + b
>>> print(a, b, c)
5 -3.4 1.6
```

### 3 Functions and Comments

In Python we can define functions using the reserved word\footnote{Reserved words are words that are used in the language that we are not at liberty to use for our own purposes. So, for example, we could not use \texttt{def} as a variable name. Another reserved word that has been introduced is \texttt{import}. There are 33 reserved words in Python.} \texttt{def} followed, on the same line, by the function name and a set of parameters or arguments enclosed in parentheses. This line is terminated by a colon. Following that line, we put the commands that define what the function does in an indented “block” of code. If we want the function to return a value, that value should be proceeded by the key word \texttt{return}. (What we mean by “return” is rather specific and will be discussed further in later material.)

As an example, assume we want a function that, using the standard notation from a math class, is given by \(f(x) = 3x(1 - x^2)\). In Python we might realize this with

```python
>>> def f(x):
...     return 3.0 * x * (1.0 - x**2)
... 
>>> f
<function f at 0x7f9d003a0f78>
```

```python
>>> f(2)
11.76
```
Note that the prompt changed from >>> to ... after the first line of the function. This is because the interpreter knows, based on def and the colon at the end of the line, that we are defining a function. We then have to provide the statement or statements that specify what the function actually does. Each statement must be indented. The amount of indentation does not matter—hitting the tab key is the simplest way to obtain the necessary indentation. When working with Python interactively, to terminate the function definition we either enter an un-indented line or enter a blank line (as was done in the example shown above). Once the blank line was entered, the prompt changed back to >>>. In this function there is a single indented line that specifies what the function does. However we could have any number of lines constituting the body of the function.

We are now free to use our function:

```python
>>> f(0.5)
1.125
>>> f(2 + 3)
-360.0
>>> a = 2
>>> b = 3
>>> c = f(a + b)
>>> c
-360.0
```

As with the mathematical functions with which you are already familiar, functions in Python can have multiple arguments. Also, in Python we can include comments in our code by using the “#” symbol. Anything after the “#” symbol is ignored by the interpreter. As an example:

```python
>>> def g(a, x):
...     # this is a comment
...     return a * x * (1.0 - x ** 2)
...     # the interpreter ignores this
... >>>
```

This defines a function $g(a, x) = ax(1 - x^2)$. Note that the “a” in this function is local to this function—it has nothing to do with any variable with the same name we may have defined elsewhere. Thus, after defining this function, this is how Python would behave:

```python
>>> a = 7  # set value of "a" for sake of illustration
>>> g(3, 0.5)
1.125
>>> g(0, 0.5)
0.0
>>> a  # show that "a" is unchanged
7
```

In the declaration of a function one must provide the parentheses which enclose the list of arguments (or parameters). However, some functions have no arguments. Nevertheless the parentheses must still be provided as shown here:

```python
>>> def big_num():
...     return 3 ** 100
... >>> 1 + big_num()
515377520732011331036461129765621272702107522002
```
The last statement shown above adds one to the result returned by the `big_num` function. Even though `big_num` takes no arguments, we also have to include parentheses when we call (or “invoke”) this function.

4 Strings

Strings are essentially collections of characters. In Python you can specify a string by enclosing the desired characters in single quotes or double quotes. If a string spans multiple lines, you repeat the initial quotation mark three times:

```python
>>> a = 'this is a string'
>>> b = "this is another string"
>>> c = '2' # because of quotes, this is a string too!
>>> d = '''this is a string
... that spans two lines'''
```

As before, to see the value of a variable, we enter the variable on a line by itself or use a `print` function:

```python
>>> a
'this is a string'
>>> d
'this is a string
that spans two lines'
>>> print(d)
this is a string
that spans two lines
```

Note that if we enter the variable on a line and hit return we are shown the way Python thinks about the string. The quotation marks are shown. Also, interestingly, when we enter `d` on a line by itself what we thought would span two lines shows up as a single line with “\n” where a line break was expected. The special characters “\n” represent the newline character which is what is embedded in the string to indicate the location of a line break. When we use the `print` function, we see the string in the expected form. Quotation marks have been removed and the newline character actually produces a newline.

Sticking with the variable declarations shown above, we can concatenate (i.e., stick together) strings

```python
>>> a_and_b = a + b
>>> print(a_and_b)
this is a stringthis is another string
```

First, we should point out that variable names and function names can consist of any collection of letters (either upper or lower case), numbers, and the underscore character but, the leading character cannot be a number. Importantly, note that in the code shown above the plus sign (+) indicates concatenation whereas a plus sign indicated addition when dealing with two numbers. No space was added between the strings being concatenated. If we wanted a space between the strings we could write

---

7This is known as “operator overloading”—the plus sign has different means depending on the context in which it is used.
Earlier we defined the variable c to be the string 2. This is not the same as the number 2 although it would be hard to tell the difference from just a print function. As an example, repeating the declaration of c shown above, we have

```python
>>> c = '2'  # quotation marks make this is a string
>>> num = 2   # this is an integer (number)
>>> print(c, num)  # print the values of c and num
2 2
>>> c, num       # display the values of c and num
('2', 2)
>>> e = c + num
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Note that trying to add a string and a number (in this case an integer or "int") produces an error. As far as Python is concerned, there is no way to add a string and a number. Also note that the command that contained just “c, num” displayed the values of these variables. When there is more than one variable on a line, the output will be enclosed in parentheses and the values will be separated by commas. The quotation marks in that output indicate c is a string.

Let us create a function called add_it() that takes two arguments and prints the “sum” of those arguments:

```python
>>> def add_it(a, b):
...   print(a + b)
```

One of the powerful features of Python is that this function is happy to accept any pair of arguments that can legitimately be “added.” For example

```python
>>> add_it(3, 27.2)
30.2
>>> add_it('this', ' and that')
this and that
```

Python's ability to handle different types of arguments in this way relates to Python's underlying object-oriented structure. For now we won't have much to say about objects and object-oriented programming, but we will explore this later.

## 5 Lists and the range() Object

Lists are sequential collections of data. (Lists are often referred to as arrays.) Up to this point all the data (or variables) we have considered have been scalar values, i.e., a single item. We can create a list by enclosing a collection of comma-separated data in square brackets. For example, here is a list of five numbers

```python
>>> a = [37.75, 1.0, -3.125, 7.0, 15.5]
```
In this case we think of the variable $a$ as pointing to a list of five elements. We can determine the length of a list using the `len()` function. To access individual elements, we specify the list we're interested in (typically this would be the variable that points to the list) followed immediately by square brackets that enclose the index of the desired element. Note that elements are indexed starting from zero! Think of this index as the offset from the start of the list rather than an ordinal number. So, the first element has index 0, the next 1, and so on until the last element of the list which has an index one less than the total length of the list:

```python
>>> len(a)  # length of list
5
>>> a[0]   # first element in list
37.75
>>> a[4]   # last element in this list
15.5
>>> a[len(a) - 1]  # another way to access last element of any list
15.5
```

The last statement above shows that it is okay to use an expression inside the brackets provided that expression returns an integer index that is valid. An invalid index is one that is outside the range of indices for the list:

```python
>>> a[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

It is possible to have an empty list (such a list is something that you encounter not infrequently—it is often the starting point for constructing a list or the end point after taking apart a list). As an example:

```python
>>> b = []  # empty list: brackets with nothing inside
>>> len(b)
0
```

Lists do not have to contain only numbers. They can contain any combination of strings, lists, numbers, etc. For example

```python
>>> a = [37.75, 1.0, -3.125, 7.0, 15.5]
>>> b = [2, (3.0 - 2.0j), 'hi!', a]
>>> b
[2, (3-2j), 'hi!', [37.75, 1.0, -3.125, 7.0, 15.5]]
```

Here the element `b[3]` is actually the list `a`. To access the individual elements of this list-within-a-list, we need another set of brackets, e.g., `b[3][2]` is -3.125.

Prior to Python 3, there was a built-in function called `range()` that returned a list of values that were a function of the arguments. This was a very simple function and yet used in numerous ways in Python code. In Python 3, the underlying implementation of `range()` changed—it is no longer a function but rather a class of objects. (We won’t concern ourselves in these notes

---

8 There are several other ways to access elements of a list that will be discussed later.

9 It might seem that negative indices are not valid, but, in fact, they can be used to count offsets from the end of the list. The use of negative indices will not be covered here.
with the distinction between functions and classes of objects.) This new implementation makes the
description of some of the details concerning range() complicated, but basically what range() ultimately allows one to do hasn’t changed significantly. At the interactive prompt you can enter “help(range)” to see what Python has to say about range(). You will see, in part, the following:

```python
>>> help(range)
Help on class range in module builtins:

class range(object)
    range([start,] stop[, step]) -> range object
    Returns an iterator that generates the numbers in the range on demand.

This indicates, perhaps somewhat cryptically, that range() can take one, two, or three arguments. If only one argument is given, it is assumed to be the “stop” value. The statement range(i) can be used to produce a list with values going from 0 to i-1 (i.e., one less than the stop value which is the single argument). To explicitly see this list, we need to enclose range() in another class object list(). So, for example, we can obtain a list of five elements with the following statement

```python
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Now, recall the list a shown above that had five elements. Here is a way to produce a list that corresponds to all the valid indices in the list a

```python
>>> list(range(len(a)))
[0, 1, 2, 3, 4]
```

This may seem rather contrived but, in the context of for loops which will be described later, a similar statement has important applications.

Using the additional (optional) arguments for range() we can generate lists with different start points, different end points, and different steps between the elements. If there are two arguments, they are assumed to be the start and stop points (and the step or increment is assumed to be one). If there are three arguments, they are assumed to correspond to the start and stop points, and the step, respectively. The following illustrates this behavior:

```python
>>> list(range(-5, 5))  # count up to (but excluding) 5 from -5
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(range(4, -1, -1))  # generate a descending list
[4, 3, 2, 1, 0]
>>> list(range(1, 20, 2))  # count by 2’s starting from 1 up to 20
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Note, as mentioned in the help description, that the “end” value is never included in the resulting list. In the following section we will see that when the range() object is used in practice, we almost never explicitly generate a list using list()—there is typically no need.
Lists are very powerful and flexible data structures and there is much that is being left unsaid here. We will explore lists in more detail later.

6 for Loops

The last topic of this introduction is the for loop. A for loop repeatedly executes a block of statements. Effectively the block of statements is executed one time for each element in a list\(^\text{10}\). Each time through the loop a variable (whose name we specify) takes on the value of one of the elements in the list. A for loop starts with the reserved word for followed by a variable name followed by the reserved word in and finally a list (or an expression that returns a list) and a colon.

As a first example, consider this code where first a list called indices is created using \texttt{range()}. This is followed by a for loop that prints the values of this list one at a time:

\begin{verbatim}
>>> # use range() to create a list called 'indices'
>>> indices = list(range(5))
>>> # let's see what 'indices' contains
>>> indices
[0, 1, 2, 3, 4]
>>> for i in indices:  # print elements of indices list
...     print(i)
...
0
1
2
3
4
\end{verbatim}

Here the loop-variable name is “i” but there is nothing unique about this name. For example, we could have called it “index” as in the following:

\begin{verbatim}
>>> for index in indices:  # print elements of indices list
...     print(index)
...
0
1
2
3
4
\end{verbatim}

We can obtain the same output by directly using \texttt{range()} in the for loop. In fact, in general we wouldn’t want to explicitly generate a list using \texttt{list()}—that can be needlessly wasteful. Thus, we can simply write

\begin{verbatim}
>>> for i in range(5):  # print elements of indices list
...     print(i)
...
\end{verbatim}

\(^{10}\text{Often that list is generated on-the-fly by an “iterator.” That is actually what range() does. But, these details aren’t important to us.}\)
As another example, let’s assume there is a list called numbers (that happens to have five elements). We can again use the range() function to create a list called indices where this list contains the valid indices for the numbers list. Then, let us write a loop that shows each element of numbers preceded by the index for that element. This code accomplishes that:

```python
>>> # create a list of numbers
>>> numbers = [37.75, 1.0, -3.125, 7.0, 15.5]
>>> # use range() to create a list called 'indices' that contains
>>> # the indices for the elements of the numbers list
>>> indices = list(range(len(numbers)));
>>> # to ensure the indices are correct, let’s inspect 'indices'
>>> indices
[0, 1, 2, 3, 4]
>>> for i in indices: # print elements of list
...     print(i, numbers[i])
...
0 37.75
1 1.0
2 -3.125
3 7.0
4 15.5
```

The first time through the loop the value of the variable i is 0 (corresponding to the first element of the list indices), the next time through i is 1, and so on, until finally, in the last iteration, i takes on the value of the last element of indices which is 4. Thus, since i is 0 the first time through the loop, the print function was effectively “print(0, numbers[0])” for this first pass. This produced the output 0, 37.75 since 37.75 was the value of the first element of the numbers list.

In practice it is unlikely that one would first create the indices list. There isn’t really any need for it. Instead, a cleaner way to obtain the same output would be with the following code:

```python
>>> # create a list of numbers
>>> numbers = [37.75, 1.0, -3.125, 7.0, 15.5]
>>> for index in range(len(numbers)): # print elements of list
...     print(index, numbers[i])
...
0 37.75
1 1.0
2 -3.125
3 7.0
4 15.5
```

[11]There is even a slightly “cleaner” way to implement this, using the enumerate() object, but we will not consider that here.
Again, in this for loop there is nothing unique about the name of the variable index. We could use any legitimate variable name.

If we were just interested in displaying the elements of the list numbers, there was really no reason to create the list we called indices—in many ways this list is a needless complication. We could instead have written simply

```python
>>> numbers = [37.75, 1.0, -3.125, 7.0, 15.5]  # list of numbers
>>> for number in numbers:  # equate number directly with elements of numbers
...     print(number)
```

This loop shows a common Python idiom: the loop variable and the name of the list have similar names where the list is a plural noun and the loop variable is a singular noun (helping to indicate that each time through the loop the loop variable takes on the value of one element from the list). In this code, similar to the first example in this section, the first time through the loop the variable number takes on the value of the first element of the list numbers, the next time through it takes on the value of the second element, and so on.

As indicated above, by directly using \texttt{range()} in a for loop statement, we can construct a loop that executes a specified number of times. The following loop executes 10 times and generates numbers in the Fibonacci sequence\footnote{There is a cleaner way to generate Fibonacci numbers using Python but it involves something that is better presented elsewhere. For now this code will suffice.}

```python
>>> old = 1  # seed the starting values
>>> new = 1
>>> for i in range(10):  # execute loop 10 times
...     print(new)  # print current ‘new’ value
...     tmp = new  # store a copy of ‘new’ value
...     new = new + old  # reset new to be sum of old and new
...     old = tmp  # set old to previous new value
```

In the Fibonacci sequence each number is the sum of the previous two numbers. Here we see that a “block” of code can span multiple lines. Provided that all these lines of code are indented the same
amount, they will all be executed each pass through the loop. Note that in this loop the variable \( i \) was never used except as a counter—it was not used in the calculation of the Fibonacci numbers. Within the loop the variable \( \text{tmp} \) was used as a temporary place to hold the value of \( \text{new} \) (since we both had to update the value of \( \text{new} \) and set \( \text{old} \) to the previous value of \( \text{new} \)).