Chapter 1

Basic Python

This book is not intended as a first course in programming. It will be assumed that the reader has some experience with programming. Therefore, it will be assumed that certain concepts are already familiar to them, the most basic of which is a mental model for programming that is sometimes called *Sequence, Selection, and Iteration*.

1.1 Sequence, Selection, and Iteration

A recurring theme in this course is the process of moving from *thinking about* code to *writing* code. We will try to shape the way we think about programs, the way we write programs, and how we go between the two in *both* directions. That is, we want to have facility with both direct manipulation of code as well as high-level description of programs.
A nice model for thinking about (imperative) programming is called Sequence-Selection-Iteration. It refers to: 1. **Sequence**: Performing operations one at a time in a specified order. 2. **Selection**: Using conditional statements such as `if` to select which operations to execute. 3. **Iteration**: Repeating some operations using loops or recursion.

In any given programming language, there are usually several mechanisms for selection and iteration, while sequencing is just the default behavior.
In fact, you usually have to have special constructions in a language to do something other than performing the given operations in the given order.

1.2 Expressions and Evaluation

Python can do simple arithmetic. For example, `2+2` is a simple arithmetic **expression**. Expressions get **evaluated** and produce a **value**. Some values are numerical like the `2 + 2` example, but they don't need to be. For example, `5 > 7` is an expression that evaluates to the boolean value `False`. Expressions can become more complex by combining many operations and functions. For example, `5 * (3 + abs(-12) / 3)` contains four different functions. Parentheses and the order of operations determine the order that the functions are evaluated. Recall that in programming the order of operations is also called **operator precedence**. In the above example, the functions are executed in the following order: `abs`, `/`, `+`, `*`.

1.3 Variables, Types, and State

Imagine you are trying to work out some elaborate math problem without a computer. It helps to have paper. You write things down, so that you can use them later. It's the same in programming. It often happens that you compute something and want to keep it until later when you will use it. We often refer to stored information as **state**.

We store information in **variables**. In Python, a variable is created by an **assignment** statement. That is a statement of the form:

```
variable_name = some_value
```

The equals sign is *doing* something (assignment) rather than *describing* something (equality). The right side of `=` is an expression that gets evaluated first. Only later does the assignment happen. If the left side of the assignment is a variable name that already exist, it is overwritten. If it doesn't already exist, it is created.

The order of evaluation is very important. Having the right side evaluated first means that assignments like `x = x + 1` make sense, because the value of `x` doesn't change until after `x + 1` is evaluated. Incidentally, there is a shorthand for this kind of update: `x += 1`. There are similar notations for `-=`, `*=`, and `/=`.

An assignment statement is not expression. It doesn't have a value. This turns out to be useful in avoiding a common bug arising from confusing assignment and equality testing (i.e. `x == y`). However, multiple assignment like `x = y = 1` does work as you would expect it to, setting both `x` and `y` to `1`.

Variables are just names. Every name is associated with some piece of data, called an object.
The name is a string of characters and it is mapped to an object. The name of a variable, by itself, is treated as an expression that evaluates to whatever object it is mapped to. This mapping of strings to objects is often depicted using boxes to represent the objects and arrows to show the mapping.

Every object has a **type**. The type often determines what you can do with the variable. The so-called **atomic types** in Python are *integers*, *floats*, and *booleans*, but any interesting program will contain variables of many other types as well. You can inspect the type of a variable using the `type()` function. In Python, the word *type* and *class* mean the same thing (most of the time).

The difference between a variable and the **object** it represents can get lost in our common speech because the variable is usually acting as the *name* of the object. There are some times when it's useful to be clear about the difference, in particular when copying objects. You might want to try some examples of copying objects from one variable to another. Does changing one of them affect the other?

```
x = 5
y = 3.2
z = True
print("x has type", type(x))
print("y has type", type(y))
print("z has type", type(z))
```

```
x has type <class 'int'>
y has type <class 'float'>
z has type <class 'bool'>
```

You should think of an object as having three things: an *identity*, a *type*, and a *value*. Its identity cannot change. It can be used to see if two objects are actually the same object with the `is` keyword. For example, consider the following code.

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y)
print(x is z)
print(x == z)
```

```
True
False
True
```

An object cannot change its identity. In Python, you also cannot change the type of an object. You can reassign a variable to point to different object of a different type, but that's not the same thing. There are several functions that may seem to be changing the types of objects, but they are really just creating a new object from the old.

```
x = 2
print("x =", x)
print("float(x) =", float(x))
print("x still has type", type(x))

print("Overwriting x.")
x = float(x)
print("Now, x has type", type(x))
```

```
x = 2
float(x) = 2.0
```

```
x still has type <class 'int'>
Overwriting x.
Now, x has type <class 'float'>
```

You can do more elaborate things as well.

```
numstring = "3.1415926"
y = float(numstring)
print("y has type", type(y))

best_number = 73
x = str(best_number)
print("x has type", type(x))

thisworks = float("inf")
print("float(\'inf\') has type", type(thisworks))
infinity_plus_one = float('inf') + 1
```

```
y has type <class 'float'>
x has type <class 'str'>
float('inf') has type <class 'float'>
```

This last example introduced a new type, that of a **string**. A string is a sequence of characters. In Python, there is no special class for a single character (as in C for example). If you want a single character, you use a string of length one.

The value of an object may or may not be changed, depending on the type of object. If the value can be changed, we say that the object is **mutable**. If it cannot be changed, we say that the object is **immutable**. For example, strings are immutable. If you want to change a string, for example, by converting it to lowercase, then you will be creating a new string.

1.4 Collections

The next five most important types in Python are strings, lists, tuples, dictionaries, and sets. We call these collections as each can be used for storing a collection of things. We will see many other examples of collections in this course.

1.4.1 Strings (`str`)

Strings are sequences of characters and can be used to store text of all kinds. Note that you can **concatenate** strings to create a new string using the plus sign. You can also access individual characters using square brackets and an **index**. The name of the class for strings is `str`. You can often turn other objects into strings.

```
s = "Hello, "
t = "World."
u = s + t
print(type(u))
print(u)
print(u[9])
n = str(9876)
print(n[2])
```

```
<class 'str'>
Hello, World.
r
7
```

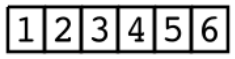
1.4.2 Lists (`list`)

Lists are ordered sequences of objects. The objects do not have to be the same type. They are indicated by square brackets and the **elements** of the list are separated by commas. You can append an item to the end of a list `L` by using the command `L.append(newitem)`. It is possible to index into a list exactly as we did with strings.

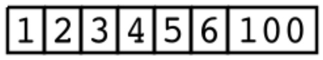
```
L = [1,2,3,4,5,6]
print(type(L))
```

```
<class 'list'>
```

Here is a common visual representation of the list.



```
L.append(100)
```



1.4.3 Tuples (`tuple`)

Tuples are also ordered sequences of objects, but unlike lists, they are immutable. You can access the items but you can't change what items are in the tuple after you create it. For example, trying to `append` raises an exception.

```
t = (1, 2, "skip a few", 99, 100)
print(type(t))
print(t)
print(t[4])
t.append(101)
```

```
<class 'tuple'>
(1, 2, 'skip a few', 99, 100)
100
```

1.4.4 Dictionaries (dict)

Dictionaries store *key-value* pairs. That is, every element of a dictionary has two parts, a **key** and a **value**. If you have the key, you can get the value. The name comes from the idea that in a real dictionary (book), a word (the key) allows you to find its definition (the value). Notice that the keys can be different types, but they must be immutable types such as atomic types, tuples, or strings. The reason for this requirement is that we will determine where to store something using the key. If the key changes, we will look in the wrong place when it's time to look it up again.

Dictionaries are also known as maps, **mappings**, or hash tables. We will go deep into how these are constructed later in the course. A dictionary doesn't have a fixed order.

```
d = dict()
d[2] = "two"
d[5] = "five"
d["pi"] = 3.1415926

print(d)
print(d["pi"])
```

```
{2: 'two', 5: 'five', 'pi': 3.1415926}
3.1415926
```

1.4.5 Sets (set)

Sets correspond to our notion of sets in math. They are collections of objects without duplicates. We use curly braces to denote them and commas to separate elements. As with dictionaries, a set has no fixed ordering. We say that sets and dictionaries are **nonsequential collections**.

Be careful that empty braces indicates an empty dictionary and not an empty set. Here is an example of a newly created set. Some items are added. Notice that the duplicates have no effect on the value as its printed.

```
s = {2,1}
print(type(s))
s.add(3)
s.add(2)
s.add(2)
s.add(2)
print(s)
```

```
<class 'set'>
{1, 2, 3}
```

1.5 Some common things to do with collections

There are several operations that can be performed on any of the collections classes (and indeed often on many other types objects).

You can find the number of elements in the collection (the **length**) using `len`.

```
a = "a string"
b = ["my", "second", "favorite", "list"]
c = (1, "tuple")
d = {'a': 'b', 'b': 2, 'c': False}
e = {1,2,3,4,4,4,4,2,2,2,1}

print(len(a), len(b), len(c), len(d), len(e))
```

```
8 4 2 3 4
```

For the sequential types (lists, tuples, and strings), you can **slice** a subsequence of indices using square brackets and a colon as in the following examples. The range of indices is half open in that the slice will start with the first index and proceed up to but not including the last index. Negative indices count backwards from the end. Leaving out the first index is the same as starting at 0. Leaving out the second index will continue the slice until the end of the sequence.

Important: slicing a sequence creates a new object. That means a big slice will do a lot of copying. It's really easy to write inefficient code this way.

```
a = "a string"
b = ["my", "second", "favorite", "list"]
c = (1, 2,3,"tuple")
print(a[3:7])
print(a[1:-2])
print(b[1:])
print(c[:2])
```

```
trin
 stri
['second', 'favorite', 'list']
(1, 2)
```

1.6 Iterating over a collection

It is very common to want to loop over a collection. The pythonic way of doing iteration is with a `for` loop.

The syntax is shown in the following examples.

```
mylist = [1,3,5]
mytuple = (1, 2, 'skip a few', 99, 100)
myset = {'a', 'b', 'z'}
mystring = 'abracadabra'
mydict = {'a': 96, 'b': 97, 'c': 98}

for item in mylist:
    print(item)

for item in mytuple:
    print(item)

for element in myset:
    print(element)

for character in mystring:
    print(character)

for key in mydict:
    print(key)

for key, value in mydict.items():
    print(key, value)

for value in mydict.values():
    print(value)
```

There is class called `range` to represent a sequence of numbers that behaves like a collection. It is often used in for loops as follows.

```
for i in range(10):
    j = 10 * i + 1
    print(j,)
```

1.7 Other Forms of Control Flow

Control flow refers to the commands in a language that affect the order or what operations are executed. The `for` loops from the previous section are classic examples if this. The other basic forms of control flow are `if` statements, `while` loops, `try` blocks, and function calls. We'll cover each briefly and refer the reader to the python documentation for more detailed info.

An `if` statement in its simplest form evaluates an expression and tries to interpret it as a boolean. This expression is referred to as a predicate. If the predicate evaluates to `True`, then a block of code is executed. Otherwise, the code is not executed. This is the *selection* of sequence, selection, and iteration. Here is an example.

```
if 3 + 3 < 7:
    print("This should be printed.")

if 2 ** 8 != 256:
    print("This should not be printed.")
```

```
This should be printed.
```

An `if` statement can also include an `else` clause. This is a second block of code that executes if the predicate evaluates to `False`.

```
if False:
    print("This is bad.")
else:
    print("This will print.")
```

```
This will print.
```

A `while` loop also has a predicate. It is evaluated at the top of a block of code. If it evaluates to `True`, then the block is executed and then it repeats. The repetition continues until the predicate evaluate to `False` or until the code reaches a `break` statement.

```
x = 1
while x < 128:
    print(x)
    x = x * 2
```

```
1
2
4
8
16
32
64
```

A `try` block is the way to catch and recover from errors while a program is running. If you have some code that may cause an error, but you don't want it to crash your program, you can put the code in a `try` block. Then, you can *catch* the error (also known as an **exception**) and deal with it. A simple example might be a case where you want to convert some number to a `float`. Many types of objects can be converted to `float`, but many cannot. If we simply try to do the conversion and it works, everything is fine. Otherwise, if there is a `ValueError`, we can do something else instead.

```
x = "not a number"
try:
    f = float(x)
except ValueError:
    print("You can't do that!")
```

```
You can't do that!
```

A function also changes the control flow. In Python, you define a function with the `def` keyword. This keyword creates an object to store the block of code. The parameters for the function are listed in parentheses after the function name. The `return` statement causes the control flow to revert back to where the function was called and determines the value of the function call.

```
def foo(x, y):
    return 8 * x + y

print(foo(2, 1))
print(foo("Na", " batman"))
```

```
17
NaNaNaNaNaNaNaNa batman
```

Notice that there is no requirement that we specify the types of objects a function expects for its arguments. This is very convenient, because it means that we can use the same function to operate on different types of objects (as in the example above). If we define a function twice, even if we change the parameters, the first will be overwritten by the second. This is exactly the same as assigning to a variable twice. The name of a function is just a name; it refers to an object (the function). Functions can be treated like any other object.

```
def foo(x):
    return x + 2

def bar(somefunction):
    return somefunction(4)

print(bar(foo))
somevariable = foo
print(bar(somevariable))
```

```
6
6
```

1.8 Modules and Imports

As we start to write more complex programs, it starts to make sense to break up the code across several files. A single `.py` file is called a module. You can import one module into another using the `import` keyword. The name of a module, by default, is the name of the file (without the `.py` extension). When we import a module, the code in that module is executed. Usually, this should be limited to defining some functions and classes, but can technically include anything. The module also has a namespace in which these functions and classes are defined.

For example, suppose we have the following files.

```
# File: twofunctions.py

def f(x):
    return 2 * x + 3

def g(x):
    return x ** 2 - 1

# File: theimporter.py
import twofunctions

def f(x):
    return x - 1

print(twofunctions.f(1)) # Will print 5
print(f(1))              # Will print 0
print(twofunctions.g(4)) # Will print 15
```

The `import` brings the module name into the current namespace. I can then use it to identify the functions from the module.

There is very little magic in an import. In some sense, it's just telling the current program about the results of another program. Because the import (usually) results in the module being executed, it's good practice to change the behavior of a script depending on whether it is being run directly, or being run as part of an import. It is possible to check by looking at the `__name__` attribute of the module. If you run the module directly (i.e. as a script), then the `__name__` variable is automatically set to `__main__`. If the module is being imported, the `__name__` defaults to the module name. This is easily seen from the following experiment.

```
# File: mymodule.py
print("The name of this module is", __name__)
```

```
The name of this module is __main__
```

```
# File: theimporter.py

import mymodule
print("Notice that it will print something different when imported?")
```

Here is how we use the `__name__` attribute to check how the program is being run.

```
def somefunction():
    print("Real important stuff here.")

if __name__ == '__main__':
    somefunction()
```

```
Real important stuff here.
```

In the preceding code, the message is printed only when the module is executed as a script. It is not printed (i.e. the `somefunction` function is not called) if the module is being imported. This is a very common python idiom.

One caveat is that modules are only executed the first time they are imported. If, for example, we import the same module twice, it will only be executed once. At that point, the namespace exists and can be accessed for the second one. This also avoids any infinite loops you might try to construct by having two modules, each of which imports the other.

There are a couple other common variations on the standard `import` statement.

1. You can `import` just a particular name or collection of names from a module: `from modulename import thethingIwanted`. This brings the new name `thethingIwanted` into the current namespace. It doesn't need to be preceded by `modulename` and a dot.

2. You can`import` all the names from the module into the current namespace: `from modulename import *`. If you do this, every name defined in the module will be accessible in the current namespace. It doesn't need to be preceded by `modulename` and a dot. Although easy to write and fast for many things, this is generally frowned upon as you often won't know exactly what names you are importing when you do this.

3. You can rename module after importing it: `import numpy as np`. This allows you to use a different name to refer to the objects of the module. In this example, I can write `np.array` instead of `numpy.array`. The most common reason to do this is to have a shorter name. The other, more fundamental use is to avoid naming conflicts.