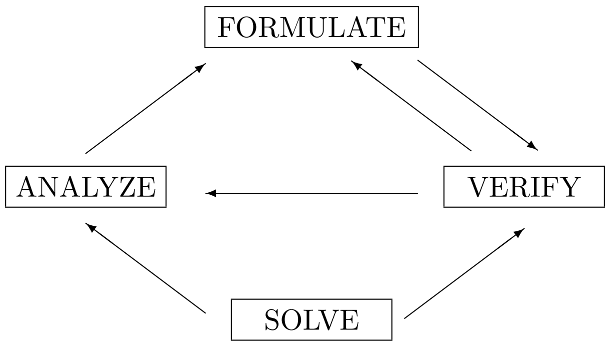
George Mason University

CDS 230

Fall 2019

Carlos Cruz



Modeling and Simulation I

Course Notes

Contents

Chapter 1

Setup

1.1 Your first program

This week, our plan is to lead you into the world of Python programming by taking you through the basic steps required to get a simple program running. The Python system (or simply Python) is a collection of applications, not unlike many of the other applications that you are accustomed to using (such as your word processor, email program, and web browser). As with any application, you need to be sure that Python is properly installed on your computer. You also need a text editor and a terminal application. By now, you should have installed the Python programming environment using the Anaconda distribution.

1.1.1 What is a Python program?

A Python program is nothing more than a sequence of characters stored in a file whose name has a *.py* extension. Python executes this sequence of statements in a specific, consistent, and predictable order. To create one, you need only define that sequence characters using a text editor.

A Python **statement** contains zero or more expressions. A statement typically has a side effect such as printing output, computing a useful value, or changing which statement is executed next.

A Python **expression** describes a computation, or operation, performed on data. For example, the arithmetic expression 2+1 describes the operation of adding 1 to 2. An expression may contain sub-expressions - the expression 2+1 contains the sub-expressions 2 and 1.

Evaluating an expression computes a Python value. This means that the Python expression 2 is different from the value 2.

The program *hello.py*, shown below, is an example of a complete Python program. The line numbers are shown to make it easy to reference specific lines, but they are not part of the program and should not be in your hello.py file.

```
1 print("Hello World!")
```

The program's sole action is to write a message back to the terminal window. A Python program consists of statements. Typically you place each statement on a distinct line.

1.1.2 Executing a Python Program

Once you compose the program, you can run (or execute) it. When you run your program the Python **compiler** translates your program into a language that is more suitable for execution on a computer[1]. Then the Python **interpreter** directs your computer to follow the instructions expressed in that language. Note that the **interpreter** is a loop[2] that:

- Reads an expression
- Evaluates the expression
- Prints the result

If the result is **None**, the interpreter does not print it. To run your program, type the python command followed by the name of the file containing the Python program in a terminal window.

```
$ python hello.py
```

For the time being, all of your programs will be just like hello.py, except with a different sequence of statements. The easiest way to compose such a program is to:

- Copy hello.py into a new file whose name is the program name followed by *.py*.
- Replace the code with a different statement or sequence of statements.

1.1.3 Python interpreter vs. Python program

Running a Python file as a program gives different results from pasting it line-by-line into the interpreter. In general the interpreter prints more output than the program would. That's because in the Python interpreter, evaluating a top-level expression prints its value while in a Python program, evaluating an expression generally does not print any output.

1.1.4 Errors

It is easy to blur the distinction among editing, compiling, and interpreting programs. You should keep them separate in your mind when you are learning to program, to better understand the effects of the errors that inevitably arise.

You can fix or avoid most errors by carefully examining the program as you create it. Some errors, known as *compile-time errors*, are raised when Python compiles the program, because they prevent the compiler from doing the translation. Python reports a compile-time error as a *SyntaxError*. Other errors, known as *run-time errors*, are not raised until Python interprets the program.

[1]Though Python is known as an interpreted language, when you run a Python **program** the source code is compiled into a much simpler form called **bytecode**. This also happens at the Python interactive prompt. However, you will never notice this compilation steps because it is implicit.

[2]An interpreter is also called a ”read-eval-print loop”, or a REPL

1.1.5 References

You are encouraged to visit the official Python website, `http://www.python.org`. More specifically:

- `http://docs.python.org/reference/index.html` provides information on the Python language.
- `http://docs.python.org/library/index.html` provides information on the Python standard libraries.
- `http://www.python.org/dev/peps/pep-0008/` provides information on Python programming style.

1.1.6 Programming Style

One final item that deserves some elaboration is programming style.

The overarching goal when composing code is to make it easy to understand. Understandable programs are more likely to be correct, and are more likely to stay correct as they are maintained over time.

Programmers use style guides to make programs easier to understand. The official Python style guide is given in `http://www.python.org/dev/peps/pep-0008/`. We recommend that you give the style guide a quick read now, and that your return to it occasionally as you gain more experience with composing Python programs.

Chapter 2

Variables and Data Types

2.1 Fundamentals

2.1.1 Operators

An operator is a symbol that represents an operation that may be performed on one or more operands. For example, the + symbol represents the operation of addition. An operand is a value that a given operator is applied to, such as operands 2 and 3 in the expression $2 + 3$.

```
An operator is a symbol that represents an operation that may be performed
on one or more operands. Operators that take one operand are called unary
operators. Operators that take two operands are called binary operators.
```

2.1.2 Summary of Python Arithmetic Operators

+ Addition	Adds values on either side of the operator.
- Subtraction	Subtracts right hand operand from left hand operand.
* Multiplication	Multiplies values on either side of the operator
/ Division	Divides left hand operand by right hand operand
// Floor Division	Returns integer part of the quotient
% modulo	Divides left hand operand by right hand operand and returns remainder
** Exponent	Performs exponential (power) calculation on operators

Python provides two forms of division. "true" division is denoted by a single slash, /. Thus, 25/10 evaluates to 2.5. Truncating division is denoted by a double slash, //, providing a truncated result based on the type of operands applied to.

2.1.3 Python Expressions

```
An expression is a combination of symbols (or single symbol) that evaluates to a
value. Expressions, most commonly, consist of a combination of operators and
```

```
operands.
```

Open up the Python interpreter and type the following **expressions**:

```
2
1 + 2
2 **12
1/-12
(72 - 32)/9*5
```

Python will happily compute their values. The first three expressions are straightforward. The fourth one would be considered very unusual or even confusing if handwritten on a piece of paper but in Python it is unambiguously correct. What about the last one? In Python an expression is evaluated from the **inside out**[1]. So, the expression $(72-32)/9*5$ is evaluated as follows:

```
(72 - 32)/9*5
(40)/9*5
40/9*5
4.44*5
22.2
```

Though this may seem trivial note what happens when you enter the following expression $(72-32)/(9*5)$? What do you get? 0.88. Well, perhaps that's what you want to compute. However, if you are trying to convert degrees Fahrenheit to degrees Celsius then the last expression (and result) is wrong. So, **precedence of operators is important in Python** and if precedence is not clear then you should use **parentheses**[2].

When Python executes the following expressions there are differences between integer arithmetic and real (floats) arithmetic that you should keep in mind (You can do this just in your interpreter and you don't need to turn anything in for this part, but pay attention to the output!)

```
5/2
5/2.0
5.0/2
7*(1/2)
7*(1/2.0)
5**2
5.0**2
5**2.0
1/3.0
```

Note that as long as one argument is a **float** all results will be floats. In the last case the final digit is rounded. Python does this for non-terminating decimal numbers, as computers cannot store infinite numbers!

[1]More generally, Python evaluates an expression by first evaluating its sub-expressions, then performing an operation on the value. Notice that each sub-expression might have its own sub-sub-expressions, so this process might repeat several times.

[2]If you remember **PEMDAS** from elementary school then it is the same for Python: (),**,*,/,+,-

2.2 Variables

Think of a variable as a container. A variable stores a value so that you can reuse it later in your program. This reduces redundancy, improves performance, and makes your code more readable. In order to use a variable, you first store a value in the variable by assigning the variable to this value. Later, you access that variable, which looks up the value you assigned to it. It is an error to access a variable that has not yet been assigned. You can reassign a variable - that is, give it a new value - any number of times.

Note that Python's concept of a variable is different from the mathematical concept of a variable. In math, a variable's value is fixed and determined by a mathematical relation. In Python, a variable is assigned a specific value at a specific point in time, and it can be reassigned to a different value later during a program's execution.

Python stores variables and their values in a structure called a **frame**. A frame contains a set of **bindings**. A binding is a relationship between a variable and its value. When a program assigns a variable, Python adds a binding for that variable to the frame (or updates its value if the variable already exists). When a program accesses a variable, Python uses the frame to find a binding for that variable.

2.2.1 Assignment statements

An assignment statement is a directive to Python to bind the variable on the left side of the = operator to the object produced by evaluating the expression on the right side. For example, when we write c = a + b, we are expressing this action: "associate the variable c with the sum of the values associated with the variables a and b."

In lecture we disscussed how one can assign values to a variable. Let's look at that in more detail. Consider the following series of statements[3]:

```
In [1]: x =  2
In [2]: print(id(x), x)
4490380384 2
```

That big number 4490380384 denotes where the data lives in the memory and it will probably be different in your computer system. What happens if we create another variable with the same value?

```
In [3]: y = 2
In [4]: print(id(y), y)
4490380384 2
```

After two consecutive assignments the *id*s of both x and y are the same implying that we are reusing the same memory location. Python does this to *optimize* memory and only so for very special cases (in the above case for **small** integers)! We will get back to these nitty-gritty details after we introduce other data types,

For now, the take home message is that "=" in an assignment statement is different than the mathematical meaning of "=". Evaluating an expression gives a new (copy of a) number, rather than changing an existing one.

[3] *id* is a Python built in function that returns the memory address used by the variable.

2.3 Built-in Data Types

2.3.1 Fundamental Types

A data type is a set of values and a set of operations defined on those values. Many data types are built into the Python language. So far, each value we have seen is a single datum, such as an integer, decimal number, or Boolean. This week we formally introduce Python's built-in data types int (for integers), float (for floating-point numbers), str (for sequences of characters) and booleans. First, we introduce an important concept: objects.

Objects

All data values in a Python program are represented by **objects** and relationships among objects. An object is an in-computer-memory representation of a value from a particular data type. Each object is characterized by its **identity**, **type**, and **value**.

- The identity uniquely identifies an object. You should think of it as the location in the computer's memory (or memory address) where the object is stored.
- The type of an object completely specifies its behavior - the set of values it might represent and the set of operations that can be performed on it.
- The value of an object is the data-type value that it represents.

Each object stores one value; for example, an object of type int can store the value 1234 or the value 99 or the value 1333. Different objects may store the same value. For example, one object of type str might store the value 'hello', and another object of type str also might store the same value 'hello'. We can apply to an object any of the operations defined by its type (and only those operations). For example, we can multiply two int objects but not two str objects.

Integers

The int data type represents integers or natural numbers. The common arithmetic operations on integers have already been introduced.

Floats

The float data type is for representing floating-point numbers, for use in scientific and commercial applications. The common arithmetic operations for integers also work with floats.

We use floating-point numbers to represent real numbers, but they are decidedly not the same as real numbers! There are infinitely many real numbers, but we can represent only a finite number of floating-point numbers in any digital computer. For example, 5.0/2.0 evaluates to 2.5 but 5.0/3.0 evaluates to 1.6666666666666667. Typically, floating-point numbers have 15-17 decimal digits of precision.

Strings

The str data type represents strings, for use in text processing. The value of a str object is a sequence of characters. You can specify a str literal by enclosing a sequence of characters in matching single quotes. You can concatenate two strings using the operator +.

```
print('hello '+'world!')
```

Converting numbers to strings for output. Python provides the built-in function str() to convert numbers to strings. Our most frequent use of the string concatenation operator is to chain together the results of a computation for output using the print function, often in conjunction with the str() function, as in this example:

```
x = 1
y = 2
print(str(x) + '+' + str(y))
```

Converting strings to numbers for input. Python also provides built-in functions to convert strings (such as the ones we type as command-line arguments) to numeric objects. We use the Python built-in functions int() and float() for this purpose. If the user types 1234 as the first command-line argument, then the code int(sys.argv[1]) evaluates to the int object whose value is 1234.

Booleans

The bool data type has just two values: True and False. The apparent simplicity is deceiving - booleans lie at the foundation of computer science. The most important operators defined for booleans are the logical operators: *and*, *or*, and *not*.

isinstance

We can use the **isinstance** function for testing types of variables:

```
isinstance(x, float)
True
```

Finally, you can do **type casting**:

```
x = 1.5
print(x, type(x))
(1.5, <type 'float'>)
x = int(x)
print(x, type(x))
(1, <type 'int'>)
```

2.4 Formatting Text and Numbers

From Newton's second law of motion one can set up a mathematical model for the motion of the ball and find that the vertical position of the ball, called y, varies with time t according to the following formula:

$$y(t) = v_0 t + \frac{1}{2} g t^2 \tag{2.1}$$

Instead of just printing the numerical value of y in our programs, we may want to write a more informative text, typically something like

```
at t= 0.6 s, the height of the ball is 1.23 m.
```

where we also have control of the number of digits (here y is accurate up to centimeters only). How can we do that? Using Python's **str.format()**. format() is a function available to string objects that provides the ability to do complex variable substitutions and value formatting.

```
The built-in format function can be used to produce a numeric string of a
given floating-point value rounded to a specific number of decimal places.
```

2.4.1 Number Formatting

The following table shows various ways to format numbers[4] using Python's str.format(), including examples for both float formatting and integer formatting.

To run examples use print(”FORMAT”.format(NUMBER)). So, to get the output of the first example, you would run:

```
print("{:.2f}".format(3.1415926));
```

Number	Format	Output	Description
3.1415926	{:.2f}	3.14	2 decimal places
2.71828	{:.0f}	3	No decimal places
-1	{:+.2f}	-1.00	2 decimal places with sign
0.25	{:.2%}	25.00%	Format percentage
1000000000	{:.2e}	1.00e+09	Exponent notation
5	{:0>2d}	05	Pad integer with zeros (left padding, width 2)

2.4.2 string.format() basics

Here are a couple of examples of basic string substitution, the {} is the placeholder for substituted variables. If no format is specified, it will insert and format as a string.

```
s1 = "Python is {}".format("a very popular language")
s2 = "CDS230 combines {} and {} elements".format("data", "science")
```

You can also use the numeric position of the variables and change them in the strings, this gives some flexibility when doing the formatting, if you make a mistake in the order you can easily correct without shuffling all the variables around.

[4]There are many more ways. These are the ones we'll use in this class. For more information see the Python documentation.

```python
s1 = " {0} is better than {1} ".format("emacs", "vim")
s2 = " {1} is better than {0} ".format("emacs", "vim")
```

Now we can format the output at the beginning of this section:

```python
t = 0.6
y = 1.23456
print("at t= {} s, the height of the ball is {:.2f} m.".format(t,y))
```

2.5 Examples

The solution to most of the exercises in this course is a Python program. To produce the solution, you first need understand the problem and what the program is supposed to do, and then you need to understand how to translate the problem description into a series of Python statements (see Appendix B). Equally important is the verification (testing) of the program. A complete solution to a programming exercises therefore consists of two parts: the program text and a demonstration that the program works correctly. Some simple programs, like the ones in the first example below, have so simple output that the verification can just be to run the program and check the output. In cases where the correctness of the output is not obvious, it is necessary to convince yourself that the result is correct. How? This can be a calculation done separately on a calculator, or one can apply the program to a special simple test with known results.

Example 1: Suppose we are to write a program for converting Fahrenheit degrees to Celsius. The solution process can be divided into three steps:

1. Establish the mathematics to be implemented. The formula to use is $C = \frac{5}{9}(F - 32)$
2. Coding of the formula in Python: C = (5/9)*(F - 32)
3. Establish a test case. For example, room temperature $F = 70$ corresponds to $C \approx 21$. We can therefore, in our new program, set $F = 70$ and check that we get $C \approx 21$.

Solution:

```python
# Convert from Fahrenheit degrees to Celsius degrees
F = 70
C = (5.0/9)*(F - 32)
print(C)
Out[]: 21.11111111111111
```

Example 2: Show that $sin^2\theta + cos^2\theta = 1$.
Solution:

```python
from math import sin, cos, pi
x = pi/4
one = sin(x)**2 + cos(x)**2
print(one)
```

Obviously this is not a mathematical proof. Instead, it is proof that all we do with computers is an approximation and limited by how numbers are represented in a computer.

Example 3: More times that we want, we find ourselves trying to figure out why our program doesn't work. So, can you find the problem(s) with the following program?

```python
a = 2; b = 1; c = 2
from math import sqrt
q = sqrt(b*b - 4*a*c)
x1 = (-b + q)/2*a
x2 = (-b - q)/2*a
print(x1, x2)
```

Upon running the program we will get the following output:

```
    1 a = 2; b = 1; c = 2
    2 from math import sqrt
>   3 q = sqrt(b*b - 4*a*c)
    4 x1 = (-b + q)/2*a
    5 x2 = (-b - q)/2*a
ValueError: math domain error
```

The Python interpreter will point you where the error is occurring and the error message says that the value is wrong. You can probably check manually and note that the value inside the square root is negative. To fix the problem you would need to be able to deal with negative roots, i.e. use complex numbers. For that you need to use the **cmath** module - which deals with complex numbers in Python. So, changing "from math import sqrt" to "from cmath import sqrt" will fix the problem. *Complex* numbers and functions can be imported using the **cmath** module.

Example 4: **Trajectory of a ball**. One can show that the the trajectory of a ball thrown at an angle θ with the horizontal ball will follow a *trajectory* $y = f(x)$ through the air, where

$$f(x) = xtan\theta - \frac{1}{2v_0^2}\frac{gx^2}{cos^2\theta} + y_0 \tag{2.2}$$

In this expression, x is a horizontal coordinate, g is the acceleration of gravity, v_0 is the magnitude of the initial velocity which makes an angle θ with the x axis, and $(0, y_0)$ is the initial position of the ball. Our programming goal is to make a program for evaluating $f(x)$. The program should write out the value of all the involved variables and what their units are.

A Solution We use the SI system and assume that v_0 is given in km/h; $g = 9.81$ m/s2; x, y, and y_0 are measured in meters; and θ in degrees. The program has naturally four parts: initialization of input data, import of functions and π from math, conversion of v_0 and θ to m/s and radians, respectively, and evaluation of $f(x)$. We choose to write out all numerical values with one decimal. The program could look like this:

```python
g = 9.81     # m/s**2
v0 = 15      # km/h
theta = 60   # degrees
x = 0.5      # m
y0 = 1       # m

```

```
print("""\
v0    = {:.1f} km/h
theta = {:d} degrees
y0    = {:.1f} m
x     = {:.1f} m\
""".format (v0, theta, y0, x)
)

from math import pi, tan, cos
# Convert v0 to m/s and theta to radians
v0 = v0/3.6
theta = theta*pi/180

y = x*tan(theta) - 1/(2*v0**2)*g*x**2/((cos(theta))**2) + y0

print('y     = {:.1f} m' .format(y))
y     = -1.8 m
```

Example 5: Age in Seconds Program
We look at the problem of calculating an individual's age in seconds. It is not feasible to determine a given person's age to the exact second. This would require knowing, to the second, when they were born. It would also involve knowing the time zone they were born in, issues of daylight savings time, consideration of leap years, and so forth. Therefore, the problem is to determine an *approximation* of age in seconds. The program will be tested against calculations of age from online resources.

So, how do we get started? We will follow the guidance from appendix B.

The Problem

The problem is to determine the approximate age of an individual in seconds within 99% accuracy of results from online resources. The program must work for dates of birth from January 1, 1900 to the present.

Problem Analysis

The fundamental computational issue for this problem is the development of an algorithm incorporating approximations for information that is impractical to utilize (time of birth to the second, daylight savings time, etc.), while producing a result that meets the required degree of accuracy.

Program Design

There is no requirement for the form in which the date of birth is to be entered. We will therefore design the program to input the date of birth as integer values. Also, the program will not perform input error checking, since we have not yet covered the programming concepts for this.

Data Description

The program needs to represent two dates, the user's date of birth, and the current date. Since each part of the date must be able to be operated on arithmetically, dates will be represented by three integers. For example, May 15, 1992 would be represented as follows:

```
year=1992
month=5
day=15
```

Algorithmic Approach The Python Standard Library module *datetime* will be used to obtain the current date. (See the Python 3 Programmers' Reference.) We consider how the calculations can be approximated without greatly affecting the accuracy of the results.

We start with the issue of leap years. Since there is a leap year once every four years (with some exceptions), we calculate the average number of seconds in a year over a four-year period that includes a leap year. Since non-leap years have 365 days, and leap years have 366, we need to compute,

```
numsecs_day = (hours per day) * (mins per hour) * (secs per minute)
numsecs_year = (days per year) * numsecs_day
avg_numsecs_year = (4 * numsecs_year) + numsecs_day) // 4
avg_numsecs_month = avgnumsecs_year // 12
```

Note that if we directly determined the number of seconds between the date of birth and current date, the months and days of each would need to be compared to see how many full months and years there were between the two. Using 1900 as a basis avoids these comparisons. Thus, the rest of our algorithm is given below.

```
numsecs_1900_to_dob = (year_birth - 1900) * avg_numsecs_year +
     (month_birth - 1) * avg_numsecs_month +
     (day_birth * numsecs_day)
numsecs_1900_to_today = (current_year - 1900) * avg_numsecs_year +
     (current_month - 1) * avg_numsecs_month +
     (current_day * numsecs_day)
age_in_secs = num_secs_1900_to_today - numsecs_1900_to_dob
```

Program Implementation and Testing First, we decide on the variables needed for the program. For date of birth, we use variables month_birth, day_birth, and year_birth. Similarly, for the current date we use variables current_month, current_day, and current_year.

```
import datetime

# Inputs
month_birth = int(input('Enter month born (1-12): '))
day_birth = int(input('Enter day born (1-31): '))
year_birth = int(input('Enter year born (4 digit): '))

# Get current time
current_month = datetime.date.today().month
```

```
current_day = datetime.date.today().day
current_year = datetime.date.today().year

# test output:
print("Input is {} {} {}:".format(month_birth, day_birth, year_birth
   ))
print("Current date is {} {} {}:".format(current_month, current_day,
    current_year))

# Main algorithm
numsecs_day = 24*60*60
numsecs_year = 365*numsecs_day

avg_numsecs_year = (4 * numsecs_year) + numsecs_day) // 4
avg_numsecs_month = avgnumsecs_year // 12

numsecs_1900_to_dob = (year_birth - 1900) * avg_numsecs_year + \
     (month_birth - 1) * avg_numsecs_month + \
     (day_birth * numsecs_day)
numsecs_1900_to_today = (current_year - 1900) * avg_numsecs_year + \
     (current_month - 1) * avg_numsecs_month + \
     (current_day * numsecs_day)
age_in_secs = numsecs_1900_to_today - numsecs_1900_to_dob
print('\n You are approximately {} seconds old'.format(age_in_secs))
```

So, how old are you? Can you test your results with those of an online program? Do you think the program above is ”good enough”?

Chapter 3

Control Flow

Control flow refers to the order that instructions are executed in a program. A **control statement** is a statement that determines the control flow of a set of instructions. There are three fundamental forms of control that programming languages provide - *sequential control*, *selection control*, and *iterative control*. Collectively a set of instructions and the control statements controlling their execution is called a **control structure**.

3.1 Boolean Expressions (Conditions)

Each value in Python has a type: int, float, string, boolean, etc. A boolean can have either the value True or the value False. In Python, certain operators compute values that are True or False.

```
An expression that computes a True or False value is called a boolean expression.
```

3.1.1 Conditional Operators

There are several conditional operators:

- < less than
- > greater than
- == equal to
- >= greater than or equal to
- <= less than or equal to
- ! = not equal to

These operators not only apply to numeric values, but to any set of values that has an ordering, such as strings. Examples:

```python
print( True and True )       # prints True
print( True and False )      # prints False
print( 3 < 4 and 10 < 12 )   # prints True
print( 3 < 4 or 12 < 10 )    # prints True
print( 4 < 3 or 12 < 10 )    # prints False
print( (4 < 3 and 12 < 10) or 7 == 7 ) # prints True
print(10 < 0 and not 10 > 2) # prints False
'Alice' < 'Bob' # prints True
```

String values are ordered based on their character encoding, which normally follows a **lexographical (dictionary) ordering**. So in the last example, 'Alice' is less than 'Bob' because the Unicode (ASCII) value for 'A' is 65, and 'B' is 66.

When we have a boolean expression like $x < 4$ Python actually computes a value. In fact, it computes a boolean value of True or False. So if x currently has the value 5, the expression $x < 4$ evaluates to the value False. It follows that you can store the results of a conditional operation in a variable:

```python
z =  1 > 2
print(z)
False
```

Notice that in mathematics, the equation $z = 1 > 2$ makes no sense. In Python, the line of code $z = 1 > 2$ is perfectly fine: compute the expression $1 > 2$, which gives False, and then assign that False value into the variable z.

Consider the following example:

```python
from math import pi, sin

print( pi )                 # prints 3.14159265359
print( sin(pi) )            # prints 1.22464679915e-16
print( sin(pi) == 0 )       # prints False.  Uh-oh!
```

The problem you see above arises because floats have limited precision. That is, Python only has an approximate value for π. The numerical computation of the *sin* function is also approximate. So, careful when using == to compare floats!

3.1.2 Logical Operators

One can operate on boolean values using logical operators. *and*, *or* and *not* are Python's logical operators that operate on boolean values and evaluate to another boolean value. Interpretation of logical expressions involving not, or, and and is straightforward when the operands are Boolean:

- *not*. Logically reverses the sense of x.
- *and*. Given **x and y** expression evaluates to True if both x and y are True, False otherwise.
- *or*. Given **x or y** expression evaluates to True if either x or y are True, False otherwise.

Caveat: Notice that if the first operand of *and* evaluates to False, we're done: we know that the result of *and* must be False, regardless of the second operand. We don't even have to look at the second operand. Python **short-circuits** if it sees that the first operand of an *and* is False; it doesn't evaluate the second operand at all!

```
In short-circuit evaluation, the second operand of Boolean operators and and or
is not evaluated if the value of the Boolean expression can be determined from
the first operand alone.
```

Finally, it is interesting to note that in Python **every object has a boolean value**. Generally one finds that:

- All integers evaluate to True, except 0 which evaluates to False
- All strings evaluate to True, except the empty string

One can use the boolean function **bool** to evaluate any Python object and check its boolean value. For example bool(True) returns True and bool(1<2) return False. Run the following examples on the interpreter and try to understand the output:

```
bool("hello")
bool(1 and 1)
bool(0 and "test")
bool(False or 1)
bool(True and 10 or not 0)
```

3.2 Conditionals

All of the programs that we have examined to this point have a simple flow of control: the statements are executed one after the other in the order given. Most programs have a more complicated structure where statements may or may not be executed depending on certain conditions (conditionals), or where groups of statements are executed multiple times (loops).

3.2.1 if Statements

In Python conditionals are known as control structures because they direct the order of execution of the statements in a program. There are various structures depending on how many conditions are being evaluated.

Unary Selection

```
if condition:                                          # HEADER
    Python code that runs iff condition is True        # CLAUSE
    Proper indentation is critical
```

This is the simplest control structure. There is one condition that, if True, evaluates the statement(s) in the clause else it does nothing. First of all note that there is a colon after the condition.

More importantly is the amount of **indentation** of each program line. In most programming languages, indentation has no affect on program logic - it is simply used to align program lines to aid readability. In Python, however, indentation is used to associate and group statements. In fact, all statements within a Python **block**[1] must same the same indentation[2]. Example:

```
y = -2
a = y < 1
if a:
    print ('a is non-zero')
```

Binary Selection:

```
if condition:
    Python code that runs iff condition is True
else:
    Python code that runs iff condition is False
    ...again, indentation is important
```

This control structure divides the flow in two depending on whether the control condition is True or False. Example[3]:

```
n = int(input('Enter a number: '))
if n % 2 == 0:
    print ("Number is even")
else:
    print ("Number is odd")
print ("Done")
```

If n % 2 is true, the first clause is executed, and the second is skipped. If n % 2 is false, the first clause is skipped and the second is executed. Either way, execution then resumes after the second clause. Both clauses are defined by indentation.

Chained if Statements: There is also syntax for branching execution based on several alternatives. For this, use one or more **elif** (short for else if) clauses. Python evaluates each expression in turn and executes the clause corresponding to the first that is true. If none of the expressions are true, and an else clause is specified, then its clause is executed[4]:

```
if condition1:
    Python code that runs iff condition1 is True
elif condition2:
    Python code that runs iff condition2 is True
elif condition3:
    Python code that runs iff condition3 is True
else:
    Python code that runs iff conditions 1-3 are False
```

[1]The usual approach taken by most programming languages is to define a syntactic device that groups multiple statements into one compound statement or block. A block is regarded syntactically as a single entity.

[2]In Python, 4 spaces is the standard

[3]In this example **input** is a Python function that prompts user for input. **input** always expects a **string** which is why n is converted to an *int* via **type casting**

[4]Note the else clause is optional.

Example:

```python
n = int(input('Enter a number: '))
if n < 0:
    print ('n is negative')
elif n > 0:
    print ('n is positive')
else:
    print ('n is zero')
```

An if statement with elif clauses uses short-circuit evaluation, analogous to what you saw with the and and or operators. Once one of the expressions is found to be true and its block is executed, none of the remaining expressions are tested.

Nested if Statements:

```python
if condition1:
    Python code that runs iff condition1 is True
else:
    Python code that runs iff condition1 is False
    if condition2:
        Python code that runs iff condition2 is True
    else:
        Python code that runs iff condition2 is False
```

Example:

```python
n = int(input('Enter a number: '))
if n > 0:
    print ('n is positive')
    if n % 2 == 0:
        print ('...and also even')
    else:
        print ('...and also odd')
else:
    if n == 0:
        print ('n is zero')
    else:
        print ('n is negative')
```

Conditionals allow us to write programs that are more interesting than ”straight-line” programs, but it is still quite limited. One way to think about the power of a class of programs is in terms of how long they can take to run. Assume one line of code takes one unit of time to execute. If a ”straight-line” program has n lines of code, it will take n units of time to execute. What about a program with selection statements? It might take less than n units of time to run but it cannot take more since each line of code is executed at most once.

A program for which maximum running time is bounded by the length of the program is said to run **constant in time**. Constant-time programs are quite limited in what they can do. The study of intrinsic difficulty of problems is the topic of **computational complexity**. We may allude to this topic a few times later in the semester.

3.3 Loops

If there is one thing computers are good for is to perform repetitive tasks. For that reason loop constructs are some of the most useful ones in programming[5] and there are two types: indefinite and definite.

```
A definite loop is a program loop in which the number of times the loop will
iterate can be determined before the loop is executed. A indefinite loop  is a
program loop in which the number of times the loop will iterate is not known
before the loop is executed.
```

3.3.1 Indefinite Iteration: while Loops

A while loop is similar to an if statement: it repeats an operation **while** a condition is true. The syntax of a while-loop looks is as follows:

```
while condition:     # HEADER
    # python code    # BODY
```

The condition is an expression that evaluates to a boolean value: either True or False. Notice that while is written in lowercase, and **there is a colon after the condition**.

The body of a while-loop is made up of the lines of code that we want to be executed multiple times and, like if-statements, indentation is critical.

We shall introduce this kind of loop through an example. The task is to generate the rows of the table of Centigrade (C) and Fahrenheit (F) values. The C value starts at -20 and is incremented by 5 as long as C $\leq$ 40. For each C value we compute the corresponding F value and write out the two temperatures. We postpone to nicely format the C and F columns of numbers and perform for simplicity a plain print C, F statement inside the loop.

```
C = -20    # starting value for C
dC = 5     # increment of C in loop
while C <= 40:     # loop heading with condition
    F = (9/5)*C + 32      # Conversion from C to F
    print(C,F)     # Result
    C = C + dC     # Increment
print('Done.')
```

The first statement whose indentation coincides with that of the while line marks the end of the loop and is executed after the loop has terminated. You are encouraged to cut-paste in the code above in a file, run it and observe what happens.

Now, let's consider the following statement:

```
    C = C + dC     # Increment
```

[5] Looping is also known as **iteration**

This is an example of a counter variable or an increment function. It is important to remember that whenever you write a while loop, you must think about an appropriate increment function[6]. Incrementing the value of a variable is frequently done in loops and so there is a special short-hand notation for this and related operations:

```python
C += dC   # equivalent to C = C + dC
C -= dC   # equivalent to C = C - dC
C *= dC   # equivalent to C = C*dC
C /= dC   # equivalent to C = C/dC
```

break Statement

Loops **iterate** over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The **break** statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

```python
while condition:
    # some code
    if condition:
        break  # breaks out of loop
```

Example: Find the first positive integer divisible by both 11 AND 12.

```python
x = 1
while True:
    if x % 11 == 0 and x % 12 == 0:
        break
    x = x + 1
print (x," is divisible by 11 and 12")
```

continue Statement

The **continue** statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

```python
while condition:
    # some code
    if condition:
        continue  # goes back to check while condition
```

Let's say we want to print all natural numbers less than 100 which are *not* multiples of 3 and 5

[6]Else you can end up with an infinite loop - which maybe bad. However, programs with infinite loops are not always bad. A robot might be intended to act forever, and the structure of the code might be an infinite loop considering and taking actions. However, unintentional infinite loops are a common programming error, and can have drastic unintended consequences, like causing the user's computer to become unresponsive while all available computation power is used running the loop

```python
x = 1
while x <=100:
    x += 1
    if x % 3 ==0 or x % 5 == 0:
        continue
        #no more code is executed, we go to the next number
    print(x, end=' ')
```

This is actually not a very elegant solution and has a "mistake". Can you think of a better way?

3.3.2 for Loops

Definite iteration loops are frequently referred to as **for** loops and exists in nearly all programming languages, including Python. The most basic for loop is a simple numeric range statement with start and end values, something like this:

```
for i = 1 to 10
    <loop body>
```

In Python the for loop is not like the type above. Instead, the for loop iterates over a collection of objects, rather than specifying numeric values or conditions, something like this:

```
for i in <collection>
    <loop body>
```

More formally, the general format of a Python for loop is the following:

```python
for <var> in <iterable>:
    <statement(s)>
```

Here <iterable> is a collection of objects[7]. The <statement(s)> in the loop body are denoted by indentation, as with all Python control structures, and are executed once for each item in <iterable>. The loop variable <var> takes on the value of the next element in <iterable> each time through the loop.

A numeric range loop isn't directly built into Python but Python provides a built-in **range** function that can be used to generate a sequence of integers that a for loop can iterate over, as shown below

```python
x = range(5)
print(x)
range(0, 5)
print(type(x))
<class 'range'>
# Then one can loop:
for i in x:
    print(x)
0
```

[7]In Python, iterable means an object can be used in iteration. If an object is iterable, it can be passed to the built-in Python function iter(), which returns something called an iterator!

```
1
2
3
4
5
```

range(<begin>, <end>, <stride>) returns an **iterable** that yields integers starting with <begin>, up to but not including <end>. If specified, <stride> indicates an amount to skip between values (analogous to the stride value used for string and list slicing).

Chapter 4

Lists and Tuples

4.1 Sequence Types

We have introduced Python's built-in data types: int, float and str. Now we introduce Python's data structures[1], specifically those known as sequence types. Sequence types are qualitatively different from numeric types (int, float) because they are compound data types - meaning they are made up of smaller pieces. Strings, of course, are made up of smaller strings, each containing one character.

4.1.1 Lists

A **list** is a **linear data structure**, meaning that its elements have a linear ordering, that can store multiple pieces of information and with a single variable name. The list name, together with a non-negative integer, called the **index**, can then be used to refer to the individual items of data. Finally, a list is a **mutable** data type which means we can change its elements.

Initialization

Lists are enclosed in square brackets ([and]). These are some examples:

```python
empty_list = []
# or
empty_list = list() # list() constructor
my_list = [2, 3, 5]
shoplist = ['apple', 'mango', 'orange', 'banana']
mixed_list = [1, 'a', 3.1416, my_list]
a = ['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']
```

Note mixed_list contains mixed types, including other lists and can contain much more[2]. The list *a* contains repeated items, showing that list elements needn't be unique. Finally, a list can contain any number of objects, from zero to as many as your computer's memory will allow.

[1]As a general rule, data structures are objects that contain a possibly large number of other objects.

[2]Lists can even contain complex objects, like functions, classes, and modules, which will be discussed later.

Accessing and Editing Lists

Every element in a list is associated with an index, which reflects the position of the element in the list. Lists in Python use zero-based indexing. Thus, all lists have index values $0 \ldots n-1$, where n is the number of elements in the list.

```
my_list = [2, 3, 5]
my_list[0]  # first element in my_list
2
my_list[1]
3
```

Python allows negative indices, which "count from the right". So, my_list[-1] gives the last element of the list my_list. my_list[-2] is the element before my_list[-1], and so forth.

Elements in lists can be deleted, and new elements can be inserted anywhere. The functionality for doing this is built into the list object and accessed by a **dot notation**[3].

```
my_list.append(7) # adds 7 to end of my_list
my_list.insert(1,0) # inserts element 0 in position 1
print(my_list)
[2, 0, 3, 5, 7]
my_list.pop() # removes last element in list
print(my_list)
[2, 0, 3, 5]
del my_list[1] # deletes second element
print(my_list)
[2, 3, 5]
```

```
# Since lists are mutable we can do the following
my_list[2] = 4
print(my_list)
[2, 3, 4]
```

Slicing

A subsequence of a sequence is called a slice and the operation that extracts a subsequence is called **slicing**. Like with indexing, we use square brackets ([ ]) as the slice operator, but instead of one integer value inside we have two, separated by a colon (:). If a is a list, the expression $a[m:n]$ returns the portion of a from index m to, but not including, index n. For example:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
primes[0:3] # yields [2, 3, 5]
primes[4:5]  # yields [11] - last index is excluded!
primes[-3:-1] # yields [23, 29]
```

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

[3]Recall that in modules we can access module functions using dot notation. A list is defined in a module and so its associated functions can also be accessed using dot notation.

```
primes[0:] # prints the entire sequence
primes[9:]  # yields [29, 31]
```

You can specify a stride - either positive or negative:

```
primes[0:6:2] # Here, 2 is the stride prints
[2, 5, 11]
primes[6:0:-2]
[17, 11, 5]
```

The syntax for reversing a list works the same way it does for strings:

```
primes[::-1]
[31, 29, 23, 19, 17, 13, 11, 7, 5, 3, 2]
```

Operators

The **in** operator returns whether a given element is contained in a sequence. Example:

```
stuff = ['this', 'that', 'these', 'those']
'this' in stuff
True
```

Notice that **in** works somewhat differently with strings. It evaluates to True if one string is a substring of another. When combined with *not* we get the obvious behavior:

```
stuff = ['this', 'that', 'these', 'those']
'python' not in stuff
True
'python' in stuff
False
```

The + operator is used to denote concatenation. Since the plus sign also denotes addition, Python determines which operation to perform based on the operand types. Thus the plus sign, +, is referred to as an **overloaded operator**. If both operands are numeric types, addition is performed. If both operands are sequence types, concatenation is performed. The same applies to $*$.

```
stuff = ['this', 'that', 'these', 'those']
stuff + ['them']
['this', 'that', 'these', 'those', 'them']
stuff*2
['this', 'that', 'these', 'those', 'this', 'that', 'these', 'those']
```

Operations min/max return the smallest/largest value of a sequence, and sum returns the sum of all the elements (when of numeric type). len() return the length of the sequence. Finally, the comparison operator, ==, returns True if the two sequences are the same length, and their corresponding elements are equal to each other.

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
stuff = ['this', 'that', 'these', 'those']
min(primes) # 2
```

```
max(primes) # 31
len(primes) # 11
len(stuff) # 4
min(stuff) # 'that'
max(stuff) # 'those'
stuff == primes # False
```

4.1.2 Iterating Over List Elements vs. List Index Values

The for statement can be applied to all sequence types, including lists.

```
# Loop over list elements
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
for n in primes:
    print(n)
```

When the elements of a list need to be accessed, but not altered, a loop variable that iterates over each list element is an appropriate approach. However, there are times when the loop variable must iterate over the *index values* of a list instead.

```
nums = [10, 20, 30, 40, 50, 60]
# Iterate over elemts of list
for k in nums:
    sum = sum + k
# Loop over index values
for k in range(len(nums)):
    sum = sum + nums[k]
```

See the difference?

There are situations in which a sequence is to be traversed while a given condition is true. In such cases, a while loop is the appropriate control structure. Let's say that we need to determine whether the value 40 occurs in list *nums* above. In this case, once the value is found, the traversal of the list is terminated.

```
nums = [10, 20, 30, 40, 50, 60]
k = 0
wanted = 40
found = false
while k < len(nums) and not found:
    if nums[k] === wanted:
        found = True
    else:
        k += 1
```

4.1.3 List Comprehensions

Because running through a list and for each element creating a new element in another list is a frequently encountered task, Python has a special compact syntax for doing this, called **list**

comprehension. The general syntax reads

```
newlist = [E(e) for e in list]
```

where $E(e)$ represents an expression involving element e. Here are some examples:

```
my_nums = [i*0.5 for i in range(10)]

Cdegrees = [-5 + i*0.5 for i in range(10)]
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]

S = [x**2 for x in range(10)]
M = [x for x in S if x % 2 == 0]


In []: my_nums
Out[]: [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]

In []: Cdegrees
Out[]: [-5.0, -4.5, -4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -0.5]

In []: Fdegrees
Out[]: [23.0, 23.9, 24.8, 25.7, 26.6, 27.5, 28.4, 29.3, 30.2, 31.1]

In []: S
Out[]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In []: M
Out[]: [0, 4, 16, 36, 64]
```

4.1.4 Traversing Multiple Lists Simultaneously

It happens quite frequently that two or more lists need to be traversed simultaneously. As an alternative to the loop over indices, Python offers a special nice syntax that can be sketched as

```
for list1, list2, list3... in zip(list1, list2, list3, ...):
    # work with element e1 from list1, element e2 from list2, etc...
```

The **zip** function turns n lists (list1, list2, list3, ...) into one list of $n-tuples$, where each $n-tuple$ (e1,e2,e3,...) has its first element (e1) from the first list (list1), the second element (e2) from the second list (list2), and so forth. The loop stops when the end of the shortest list is reached.

4.1.5 Tuples

Python provides another sequence type that is an ordered collection of objects, called a tuple. Tuples are identical to lists in all respects, except for the following properties:

- Tuples are defined by enclosing the elements in parentheses (()) instead of square brackets ([]).

- Tuples are immutable.

Examples:

```
empty_tuple = ()
# or
empty_list = tuple() # list() constructor
my_tuple = (2, 3, 5, 'cat', 'dog')
```

Note: Even though tuples are defined using parentheses, you still index and slice tuples using square brackets, just as for strings and lists. Everything about lists - they are ordered, they can contain arbitrary objects, they can be indexed and sliced, they can be nested - is true of tuples as well. But they can't be modified.

Why use a tuple instead of a list?

- Program execution is faster when manipulating a tuple than it is for the equivalent list.
- Sometimes you don't want data to be modified. If the values in the collection are meant to remain constant for the life of the program, using a tuple instead of a list guards against accidental modification.
- There is another Python data type that you will encounter shortly called a dictionary, which requires as one of its components a value that is of an immutable type. A tuple can be used for this purpose, whereas a list can't be.

4.1.6 Nested Data Structures

Lists and tuples can contain elements of any type, including other sequences. Thus, lists and tuples can be nested to create arbitrarily complex data structures. Examples:

```
class_grades = [ [85, 91, 89], [78, 81, 86], [62, 75, 77]]
mixed_nested_list = [ [4, [True, False], 6, 8], [888.0, 999.0] ]
tuple_with_list = (1, [2, 3], 4, 5)
```

4.1.7 Copying Lists

Because of the way that lists are represented in Python, when a variable is assigned to another variable holding a list, $list2 = list1$, each variable ends up referring to the same *instance* of the list in memory. This is a referred to as a **shallow copy** and has important implications. For example, if an element of $list1$ is changed, then the corresponding element of $list2$ will change as well,

```
list1 = [10, 20, 30, 40]
list2 5 list1
list1[0] = 5
print(list1)
[5, 20, 30, 40] # change made in list1
print(list2)
[5, 20, 30, 40] # change in list1 causes a change in list2
```

Knowing that variables *list*1 and *list*2 refer to the same list explains this behavior. This issue does not apply to strings and tuples, since they are immutable and therefore cannot be modified. When needed, a copy of a list can be made as given below,

```
list1 = [10, 20, 30, 40]
list2 = list(list1)
list1[0] = 5
print(list1)
[5, 20, 30, 40] # change made in list1
print(list2)
[10, 20, 30, 40] # change in list1 does NOT cause a change in list2
```

When copying lists that have sublists, another means of copying, called **deep copy**, may be needed.

`When a variable is assigned to another variable holding a list, each variable ends up referring to the same instance of the list in memory.`

Python has a module called *copy* to perform these types of copy operations.

```
# importing "copy" for copy operations
import copy
list1 = [1, 2, [3,5], 4]
# using copy to shallow copy
shallow_copy = copy.copy(list1)
# using deepcopy to deep copy
deep_copy = copy.deepcopy(list1)
```

Chapter 5

Dictionaries and Sets

5.1 Sets

A mathematical set is a collection of values without duplicates or order. In sets

- Order does not matter, i.e. { 1, 2, 3 } == { 3, 2, 1 }
- There are no duplicate entries: { 3, 1, 4, 1, 5 } == { 5, 4, 3, 1 }

```
A Python set is an unordered collection of unique and immutable objects.
```

5.1.1 Properties

- Set elements must be immutable values
- The set itself is mutable (e.g. we can add and remove elements)
- Only set operations change the set
- **Aside**: *frozenset* must contain immutable values and is itself immutable (cannot add and remove elements)

5.1.2 Initialization

Sets are enclosed in parentheses. There are two ways to initialize them.
Direct mathematical syntax:

```
odd = {1,3,5}
```

Unfortunately you cannot use {} to express empty set.
Construct from a list (or a tuple or a string):

```
empty = set() # constructor
odd = set([1,3,5])
primes = set([2,3,5])
```

5.1.3 Set operations

Given the sets *odd* and *primes* above we can easily understand set operations: Note that we can

membership $\in$	Python:**in**	**4 in primes** $\Rightarrow$ False
union $\cup$	Python:|	**odd** | **primes** $\Rightarrow$ {1,2,3,5}
intersection $\cap$	Python:**&**	**odd & primes** $\Rightarrow$ {3,5}
difference $-$	Python:**-**	**odd - primes** $\Rightarrow$ {1}
symmetric difference	Python:ˆ	**odd ˆ primes** $\Rightarrow$ {1,2}

do iteration over stes

```
for n in primes:
    print(n)
```

But we cannot index into a set to access a specific element

```
print[1] # illegal
```

5.1.4 Modifying a Set

Add element to a set

```
primes.add(7)
# or
primes = primes | {7}
```

Remove element from a set

```
primes.remove(2)
# or
primes = primes - {2}
```

Remove an arbitrary element from a set

```
primes.pop()
```

Try the following:

```
z = {5, 6, 7, 8}
y = {1, 2, 3, "foo", 1, 5}
k = z & y
j = z | y
m = y - z
n = z - y
p = z
q = set(z)
z.add(9)
```

Example: Find the common element in both list1 and list2.

```python
# Using lists and loops:
list1 = [1,2,3,4,5,6,7,8]
list2 = [2,4,6,8]
out1 = []
for i in list2:
    if i in list1:
        out1 .append(i)

# Using list comprehensions would reduce the last 4 lines into 1:
out1 = [i for i in list2 if i in list1]
```

Using sets you would perform the following operation: set1 & set2:

```python
list1 = [1,2,3,4,5,6,7,8]
list2 = [2,4,6,8]
out1 = set(list1) & set(list2)
```

How would you find elements in either list1 or list2 (or both) (without duplicates)? set1 | set2
How would you find elements in either list1 or list2 but not in both? set1 ˆ set2

5.2 Dictionaries

So far in the book we have stored information in various types of objects, such as numbers, strings, list, and arrays. A dictionary is a very flexible object for storing various kind of information[1].

```
A dictionary is a mapping that stores keys with associated values.
For every key in a dictionary, there is exactly one value associated with it.
```

Recall that a list is a collection of objects indexed by an integer going from 0 to the number of elements minus one. Instead of looking up an element through an integer index, it can be more handy to use a text. Roughly speaking, a list where the index can be a text is called a dictionary in Python. Other computer languages use other names for the same thing: HashMap, hash, associative array, or map.

5.2.1 Properties

- Dictionaries are mutable objects.
- Order does not matter.
- Given a key, can look up a value. Given a value, cannot lookup its key.
- There are no duplicate keys but or or more keys may map to the same value.
- Keys must be immutable

[1]And in particular when reading files, to be discussed later.

5.2.2 Initialization

Dictionaries are enclosed in curly braces {}. There are two ways to initialize them.

```
d = {}
d = dict()  # constructor
```

Examples of dict() syntax:

```
# Temperatures:
daily_temps = {'sun': 68.8,
               'mon': 70.2,
               'tue': 67.2,
               'wed': 71.8,
               'thur': 73.2,
               'fri': 75.6,
               'sat': 74.0}
# Population of cities (in millions)
population = {"Chicago":2.7, "New York":8.17, "Rome":2.87,
                 "Paris":2.24, "London":8.78}
# HTTP response status codes
status = {200:"ok", 404:"not found", 400:"bad request"}
```

Note that (key,value) pairs are separated by a colon(:).

5.2.3 Dictionary Operations

Let's consider the first example above. Dictionary daily_temps stores the average temperature for each day of the week. Each temperature has associated with it a unique key value ('sun', 'mon', etc.). Strings are often used as key values. The syntax for accessing an element of a dictionary is the same as for accessing elements of sequence types, except that a key value is used within the square brackets instead of an index value: daily_temps['sun'].

```
Python dictionaries use index-like notation to refer to the value associated
with key in a dictionary d: d[key]
```

Although the elements of the dictionary data structure are physically ordered, the ordering is irrelevant to the way that the structure is utilized. *The location that an element is stored in and retrieved from within such a data structure depends only on its key value*, thus there is no logical first element, second element, and so forth[2]. Let's see how we can use daily_temps:

```
if daily_temps['sun'] > daily_temps['sat']:
    print('Sunday was the warmer weekend day')
else:
    if daily_temps['sun'] < daily_temps['sat']:
        print('Saturday was the warmer weekend day')
    else:
         print('Saturday and Sunday were equally warm')
```

[2]The specific location that a value is stored is determined by a particular method of converting key values into index values called **hashing**.

Although strings are often used as key values, any immutable type may be used as well, such as a tuple. In this case, the temperature for a specific date is retrieved by,

```
temps[('Feb', 5, 2019)] -> 70.0
```

Note that this key contains both string and integer values.

What follows is a summary some of the basic operations for a dictionary d.

d[key]	Get value for key in dictionary.
d[key] = value	Set value for key in dictionary d to be value.
len(d)	Number of key-value pairs in d.
key in d	True if key has an entry in d; otherwise, False.
key not in d	True if key does not have an entry in d; otherwise, False.
del d[key]	Delete entry for key in d. Raises KeyError if key is not in d.
sorted(d)	Return sorted list of keys in d. Use sorted(d, key=d.get) to sort the keys by value.

Accessing dictionary elements:

```
atomic_number = {"H":1, "Fe":26, "Au":79}
atomic_number["Au"] # prints 79
atomic_number["B"] # prints KeyError
```

Dictionary functions:

```
atomic_number = {"H":1, "Fe":26, "Au":79}
atomic_number.keys() # prints dict_keys(['H', 'Fe', 'Au'])
atomic_number.values() # prints [1, 79, 26]
atomic_number.items() # print dict_items([('H', 1), ('Fe', 26), ('Au
   ', 79)])
```

Iterating through a dictionary:

```
atomic_number = {"H":1, "Fe":26, "Au":79}

# Print out all the keys:
for element_name in atomic_number.keys():
    print(element_name)
# Print out all the values:
for element_number in atomic_number.values():
    print element_number
# Print out the keys and the values
for (element_name, element_number) in atomic_number.items():
    print("name:", element_name,"number:", element_number)
```

Modifying a dictionary:

```
us_wars1 = {
    "Revolutionary" : [1775, 1783],
    "Mexican" : [1846, 1848],
    "Civil" : [1861, 1865] }
us_wars1["WWI"] = [1917, 1918]  # add mapping
del us_wars1["Civil"]  # remove mapping
```

Example:
Reverse key with value in a dictionary: E.g. Given {5:25, 6:36, 7:49}, produce {25:5, 36:6, 49:7}

```
d = {5:25, 6:36, 7:49}
k ={}
for i in d.keys():
    k[d[i]] = i
print(k)
```

Try these on your own:

```
squares = { 1:1, 2:4, 3:9, 4:16 }
squares[3] + squares[3]
squares[3 + 3]
squares[2] + squares[2]
squares[2 + 2]
```

Chapter 6

Functions

So far, we have limited ourselves to using only the most fundamental features of Python - variables, expressions, control structures, and data structures. In theory, these are the only instructions needed to write any program (that is, to perform any computation). From a practical point-of-view, however, these instructions alone are not enough.

The problem is one of complexity. In order to manage the complexity of a large problem, it is broken down into smaller subproblems. Then, each subproblem can be focused on and solved separately. In programming, we do the same thing. Programs are divided into manageable pieces called program routines (or simply routines). Doing so is a form of *abstraction* in which a more general, less detailed view of a system can be achieved. In addition, program routines provide the opportunity for code reuse, so that systems do not have to be created from ”scratch”. Routines, therefore, are a fundamental building block in software development.

6.1 Functions

In Python program routines are called **functions**. Python functions are similar to the mathematical functions that you are familiar with but they are much more. A function is a collection of statements that you can execute wherever and whenever you want in the program. You may send variables to the function to influence what is getting computed by statements in the function, and the function may return new objects. In particular, functions help to avoid duplicating code snippets by putting all similar snippets in a common place[1]. This strategy saves typing and makes it easier to change the program later. Functions are also often used to just split a long program into smaller, more manageable pieces, so the program and your own thinking about it become clearer. Python comes with lots of functions[2] and we have met several so far.

```python
# Built-in functions
print('hi!')
len("hello")
pow(2,3)
str(17)
```

[1]Also known as the Don’t Repeat Yourself (DRY principle).

[2]Built-in functions are always available an do not need to be imported. Other functions come from modules, like **math** or **random**.

```
int(input("enter integer: "))
range(1,5)

# functions from modules
import math
math.sin(math.pi)
import random
random.random() # this one needs no input
```

6.1.1 Defining Functions

In addition to the built-in or imported functions of Python, there is the capability to define new functions and that makes programming so much more exciting.

```
A Python function is a named group of statements that accomplishes some task.
```

The elements of a function definition are:

```
def a_good_name(arg1, arg2, arg3...):    # HEADER
    # Python statements
    return value(s)   # RETURN VALUE(S)
```

The first line of a function definition is the function header. A function header starts with the keyword **def**, followed by an identifier, which is the function's name[3]. The function name is followed by a comma-separated (possibly empty) list of **identifiers** ($arg1, arg2, arg3$...) called **parameters**. The actual values passed to the function are called **arguments**. Following the parameter list is a colon (:). Following the function header is the body of the function or program block containing the function's instructions. As with all blocks, the statements must be indented at the same level, relative to the function header.

Value-Returning Functions

A value-returning function is one called for its return value, and is therefore similar to a mathematical function. Take the simple mathematical function $f(x) = 2x$. In this notation, x stands for any numeric value that function f may be applied to, for example, $f(2) = 2x = 4$:

```
def f(x): # x is a parameter
    return 2*x
```

Note there is a keyword **return** to specify what result to return. After **return** we have an expression that holds the returned value.

Of course Python functions may return more than one value. So, if we are interested in writing a function that returns the roots of a quadratic equation then we will need to return two values.

[3]Function names are important. Be careful of very short names: f(x) is almost always too vague and eventually you will have a hard time knowing or remembering what it does.

Non-Value-Returning Functions

A non-value-returning function is called not for a returned value, but for its **side effects**. A side effect is an action other than returning a function value, such as displaying output on the screen.

```
def print_hello():
    print("Hello, world")
```

Note that print_hello() has an empty parameter list and no return statement.

6.1.2 Calling Functions

Every function has the capacity to perform a task, but it only performs that task when it is called. A function call requests execution of the function with particular arguments passed as the values for its parameters (if any). The syntax of a function call is:

```
output = name(arg1, arg2, ...)
```

When this expression appears in a program statement that is being executed, the function called *name* executes, using the argument values inside parentheses as the values of its parameters and returns the values specified inside the function: *output* above must match the number of values returned by the functions. Examples:

```
def square(x):  # x is a parameter
    return x * x
sq = square(2)    # 2 is an argument, function returns sq=4

def my_sum(x, y):
    return x + y
sum = my_sum(2,3)  # returns sum=5

def quadratic_roots(a, b, c):
     # some code to solve quadratic formula
    return x1, x2
root1, root2 = quadratic_roots(1, -1, 1)
```

Note that in the last function, quadratic_roots, we need two values on the left-hand side of the assignment operator because the function returns two values

Note that there is a fundamental difference in the way that value-returning and non-value-returning functions are called:

```
def print_hello():
    print("Hello, world")

# calling print_hello()
print_hello()
```

print_hello() does not return a value and so there is no left-hand-side.

Docstrings

There is a convention in Python to insert a documentation string right after the def line of the function definition. The documentation string, known as a doc string, should contain a short description of the purpose of the function and explain what the different arguments and return values are. Interactive sessions from a Python shell are also common to illustrate how the code is used. Doc strings are usually enclosed in triple double quotes (three double-quote " characters), which allow the string to span several lines.

```
def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = a*x + b that goes
    through two points (x0, y0) and (x1, y1).
    x0, y0: a point on the line (floats).
    x1, y1: another point on the line (floats).
    return: coefficients a, b (floats) for the line (y=a*x+b). """
    a = (y1 - y0)/float(x1 - x0)
    b = y0 - a*x0
    return a, b
```

Note that the doc string must appear before any statement in the function body.

The doc string can be accessed in a code as funcname.__doc__, where funcname is the name of the function, e.g.,

```
print(line.__doc__)
```

The print() above will print out the documentation of the line() function.

6.1.3 Parameter Passing

Parameter passing is the process of passing arguments to a function. As we have seen, actual arguments are the values passed to a function's formal parameters to be operated on. For example, consider the following program (you should run it):

```
def ordered(n1, n2):  # n1, n2 are parameters
    return n1 < n2  # returns either True or False

num1 = int(input('Enter num1: '))
num2 = int(input('Enter num2: '))

if ordered(num1, num2):  # num1, num2 are arguments
    print('First number is smaller')
else:
    if ordered(num2, num1):
        print('First number is larger')
    else:
        print('Numbers are equal')
```

In this example, function ordered is called once with arguments num1, num2 and a second time with arguments num2, num1. There is one important observation:

```
The correspondence of arguments and parameters is determined by the order of the
arguments passed, and not their names.
```

Keyword Arguments

The functions we have looked at so far were called with a fixed number of **positional arguments**. A positional argument is an argument that is assigned to a particular parameter based on its position in the argument list, as shown below.

```
def mortgage_rate(amount, rate, term):
    # some calculations
    return some_value
monthly_payment = mortgage_rate(350000, 0.06, 20)
```

This function computes and returns the monthly mortgage payment for a given loan amount (amount), interest rate (rate), and number of years of the loan (term).

Python provides the option of calling any function by the use of **keyword arguments**. A keyword argument is an argument that is specified by parameter name, rather than as a positional argument as shown below:

```
monthly_payment = mortgage_rate(rate=0.06, term=20, amount=350000)
```

This can be a useful way of calling a function if it is easier to remember the parameter names than it is to remember their order. It is possible to call a function with the use of both positional and keyword arguments. However, all positional arguments must come before all keyword arguments in the function call, as shown below:

```
monthly_payment = mortgage_rate(350000, rate=0.06, term=20)
```

This form of function call might be useful, for example, if you remember that the first argument is the loan amount, but you are not sure of the order of the last two arguments rate and term.

Default Arguments

Python also provides the ability to assign a default value to any function parameter allowing for the use of **default arguments**. A default argument is an argument that can be optionally provided, as shown here:

```
def mortgage_rate(amount, rate, term=20):
    # some calculations
    return some_value
monthly_payment = mortgage_rate(350000, 0.06)
```

In this case, the third argument in calls to function mortgage_rate is optional. If omitted, parameter term will default to the value 20 (years) as shown. If, on the other hand, a third argument is provided, the value passed replaces the default parameter value.

Now try the following making sure you understand how the program works:

```
def addup(first, last, incr=-1):
    if first > last:
        sum = -1
    else:
        sum = 0
        for i in range(first, last+1, incr):
            sum = sum + i
    return sum
addup(1, 10)
addup(1, 10, 2)
addup(first=-1, last = -10)
addup(incr=-2, first=-1, last = -10)
```

6.1.4 Variable Scope

A **local variable** is a variable that is only accessible from within a given function. Such variables are said to have **local scope**. In Python, any variable assigned a value in a function becomes a local variable of the function. Consider the following

```
def func1():
    n = 10
    print('func1 ',n)
def func2():
    n = 20
    print('func2 ',n)
    func1()
    print('func2 ',n)
In [1]: func2()
func2  20
func1  10
func2  20
```

Both func1 and func2 contain identifier n. Function func1 assigns n to 10, while function func2 assigns n to 20. Both functions display the value of n when called - func2 displays the value of n both before and after its call to func1. If identifier n represents the same variable, then shouldn't its value change to 10 after the call to func1? However, as shown by the output, the value of n remains 20. This is because there are two distinct instances of variable n, each local to the function assigned in and inaccessible from the other.

Now try commenting out $n = 10$ in func1() and re-run the above. In that case you will get an error indicating that variable n is not defined within func1. This is because variable n defined in func2 is inaccessible from func1. (In this case, n is expected to be a **global variable**).

The period of time that a variable exists is called its **lifetime**. Local variables are automatically created (allocated memory) when a function is called, and destroyed (deallocated) when the function terminates. Thus, the lifetime of a local variable is equal to the duration of its function?s execution. Consequently, the values of local variables are not retained from one function call to the next.

The concept of a local variable is an important one in programming. It allows variables to be defined in a function without regard to the variable names used in other functions of the program. It also allows previously written functions to be easily incorporated into a program. The use of global variables, on the other hand, brings potential havoc to programs[4].

6.1.5 Functions as Arguments to Functions

One frequently needs to have functions as arguments in other functions. For example, for a mathematical function $f(x)$ we can have Python functions for

1. numerical root finding: solve $f(x) = 0$ approximately
2. numerical differentiation: compute $f'(x) = 0$ approximately
3. numerical integration: compute $\int_a^b f(x)dx$ approximately
4. numerical solution of differential equations: $\frac{dx}{dy} = f(x)$

In such Python functions we need to have the $f(x)$ function as an argument f. For example, consider a function for computing the second-order derivative of a function $f(x)$ numerically:

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \tag{6.1}$$

where h is a small number. The approximation 6.1 becomes exact in the limit $h \to 0$. A Python function for f"(x) can be implemented as follows:

```python
def diff2(f, x, h=1E-6):
    """
    approximation of the second-order derivative of a function
    """
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```

The f argument is like any other argument, i.e., a name for an object, here a function object that we can call as we normally call function objects. An application of $diff2$ can read

```python
# This is the function that we want to take the derivative of
def g(t):
    return t**3
t = 1.0
d2g = diff2(g, t)
print("g({})={:.8f}".format(t, d2g))
# prints
g(1.0)=5.9996   # Note the exact value is 6.0
```

Now, we know that the approximation formula 6.1 becomes more accurate as h decreases. Let's try to show this:

[4]For this reason, the use of global variables is generally considered to be bad programming style.

```
for k in range(1,15):
    h = 10**(-k)    # we will decrease the value of h
    d2g = diff2(g, 1, h)
    print("h = {}, g({})={:.8f}".format(h, t, d2g))
# prints:
h = 0.1, g(1.0)=6.00000000
h = 0.01, g(1.0)=6.00000000
h = 0.001, g(1.0)=6.00000000
h = 0.0001, g(1.0)=5.99999999
h = 1e-05, g(1.0)=6.00000272
h = 1e-06, g(1.0)=5.99964523
h = 1e-07, g(1.0)=6.03961325
h = 1e-08, g(1.0)=2.22044605
h = 1e-09, g(1.0)=444.08920985
h = 1e-10, g(1.0)=0.00000000
h = 1e-11, g(1.0)=0.00000000
h = 1e-12, g(1.0)=444089209.85006267
h = 1e-13, g(1.0)=-44408920985.00625610
h = 1e-14, g(1.0)=0.00000000
```

Note that for $h < 10^{-8}$ the answers are totally wrong. The problem is that for small h on a computer, round-off errors in the formula 6.1 blow up and destroy the accuracy. The mathematical result that 6.1 becomes an increasingly better approximation as h gets smaller and smaller does not hold on a computer! Or more precisely, the result holds until h in the present case reaches 10^{-6}.

The reason for the inaccuracy is that the numerator in 6.1 contains subtraction of quantities that are almost equal. The result is a very small and inaccurate number. The inaccuracy is magnified by h^{-2}, a number that becomes large for small h. Switching from the standard floating-point numbers (float) to numbers with arbitrary high precision resolves the problem. Python has a module **decimal** that can be used for this purpose. We may come back to explore these issues later on during this course.

6.1.6 The Main Program

In programs containing functions we often refer to a part of the pro- gram that is called the main program. This is the collection of all the statements outside the functions, plus the definition of all functions. Let us look at a complete program:

```
from math import exp, sin, pi   # in main
def f(x):                       # in main
    e = exp(-0.1*x)
    s = sin(6*pi*x)
    return e*s
x= 2                            # in main
y= f(x)                         # in main
print('f({:.4f})={:.4f}'.format(x, y))    # in main
```

The main program here consists of the lines with a comment in main. The execution always starts with the first line in the main program. When a function is encountered, its statements are just used to define the function - nothing gets computed inside the function before we explicitly call the function, either from the main program or from another function. All variables initialized in the main program become **global variables**.

6.1.7 Lambda Functions

There is a quick one-line construction of functions that is sometimes convenient:

```
f = lambda x: x**2 + 4
```

This so-called **lambda** function is equivalent to writing

```
def f:
     return x**2 + 4
```

In general,

```
def g(arg1, arg2, arg3, ...):
    return expression
```

can be written as

```
g = lambda arg1, arg2, arg3, ...: expression
```

Lambda functions are usually used to quickly define a function as argument to another function. Because lambda functions can be defined ”on the fly” and thereby save typing of a separate function with *def* and an intended block, lambda functions are popular among many programmers.

6.2 A Bioinformatics Example

The genetic code of all living organisms are represented by a long sequence of simple molecules called nucleotides, or bases, which makes up the Deoxyribonucleic acid, better known as DNA. There are only four such nucleotides, and the entire genetic code of a human can be seen as a simple, though 3 billion long, string of the letters A, C, G, and T. Analyzing DNA data to gain increased biological understanding is much about searching in (long) strings for certain string patterns involving the letters A, C, G, and T. This is an integral part of bioinformatics, a scientific discipline addressing the use of computers to search for, explore, and use information about genes, nucleic acids, and proteins.

6.2.1 Counting Letters in DNA Strings

Given some string **dna** containing the letters A, C, G, or T, representing the bases that make up DNA, we ask the question: how many times does a certain base occur in the DNA string? For example, if **dna** is **ATGGCATTA** then we ask how many times the base **A** occurs in this string. In the simple case the answer is 3.

How can we implement this in Python? The most straightforward solution is to loop over the letters in the string, test if the current letter equals the desired one, and if so, increase a counter. Looping over the letters is obvious if the letters are stored in a list. This is easily done by converting a string to a list:

```
def count_str0(dna, base):
    dna = list(dna) # convert string to list of letters
    i = 0 # counter
    for c in dna:
        if c == base:
            i += 1
    return i
dna = "ATGCGGACCTAT"
base = "C"
count = count_str0(dna , base)
print("{} appears {:d} times in {}".format(base, n, dna))
```

NOTE It is fundamental for correct programming to understand how to simulate a program by hand, statement by statement. Two tools are effective for helping you reach the required understanding of performing a simulation by hand: (i) printing variables, (ii) using a debugger.[5] You may have noticed that converting the string **dna** to a list is actually unnecessary as we can just iterate over the string (after all it is a sequence):

```
def count_str1(dna, base):
    i = 0 # counter
    for c in dna:
        if c == base:
            i += 1
    return i
dna = "ATGCGGACCTAT"
base = "C"
count = count_str1(dna , base)
print("{} appears {:d} times in {}".format(base, n, dna))
```

The same problem can be solved using a for loop to iterate over the DNA string:

```
def count_str2(dna, base):
    i = 0 # counter
    for j in range(len(dna)):
        if dna[j] == base:
            i += 1
    return i
dna = "ATGCGGACCTAT"
base = "C"
count = count_str2(dna , base)
print("{} appears {:d} times in {}".format(base, count, dna))
```

Do you see the difference with the earlier solution? Can you think of another way to solve the problem (perhaps using a while loop)?

[5]The Python Online Tutor at `http://people.csail.mit.edu/pgbovine/python/tutor.html` is, at least for small programs, a splendid alternative to debuggers.

Note that a common theme in the count_str algorithms is that we need to check when the letter we search for is found in the DNA string. Thus, the idea could be to create a list *found* where each element is True if the base is found in the DNA string. The number of True values, i.e. the length of the list, is the number of letters of the base in the DNA. Consider the following:

```python
def count_str3(dna, base):
    found = [] # matches for base in dna: found[i]=True if dna[i]==
       base for c in dna:
    for c in dna:
        if c == base:
            found.append(True)
        else:
            found.append(False)
    return sum(found) # note we are using sum here
dna = "ATGCGGACCTAT"
base = "C"
count = count_str3(dna , base)
print("{} appears {:d} times in {}".format(base, count, dna))
```

Finally we can use boolean values directly, as follows:

```python
def count_str4(dna, base):
    found = []
    for c in dna:
        found.append(c == base)
    return sum(found) # note we are using sum here
dna = "ATGCGGACCTAT"
base = "C"
count = count_str4(dna , base)
print("{} appears {:d} times in {}".format(base, count, dna))
```

Finally, let's try Python's library.

```python
def count_str5(dna, base):
    return dna.count(base)
dna = "ATGCGGACCTAT"
base = "C"
count = count_str4(dna , base)
print("{} appears {:d} times in {}".format(base, count, dna))
```

There are probably a few other ways to solve this problem. There are two lessons here:

- There may be multiple ways, i.e. algorithms, to solve a problem.
- Depending on the problem, you may want to choose the most efficient algorithm.

Deciding what constitutes an efficient algorithm is beyond the scope of this course. However, we can explore one measure of efficiency: the CPU time, i.e. which one of the above implementations is the fastest? To answer the question we need some test data, which should be a huge string of DNA.

Generating Random DNA Strings. The simplest way of generating a long string is to repeat a character a large number of times:

```
N=1000000
dna = 'A'*N
```

The resulting string is just AAA...A, of length N, which is fine for testing the efficiency of Python functions. Nevertheless, it is more exciting to work with a DNA string with letters from the whole alphabet A, C, G, and T. To make a DNA string with a random composition of the letters we can first make a list of random letters and then join all those letters to a string:

```
import random

def generate_string(N, alphabet='ACGT'):
    my_list = []
    for i in range(N):
        my_list.append(random.choice(alphabet))
    return ''.join(my_list)  # this returns a string
# test
print(generate_string(10)) # will be "random"
dna = generate_string(10000000) # 1e7 letters
```

In the snippet above, the *random.choice*(*alphabet*) function selects an element in the list *alphabet* at random. The *join* function is used to join the elements of the list into a string (see help(dna.join) for more information).

Measuring CPU Time. Our next goal is to see how much time the various count_* functions spend on counting letters in a huge string, which is to be generated as shown above. Measuring the time spent in a program can be done by the time module:

```
import time
...
t0 = time.clock()
# do stuff
t1 = time.clock()
cpu_time = t1 - t0
```

The time.clock() function returns the CPU time spent in the program since its start. If the interest is in the total time, also including reading and writing files, time.time() is the appropriate function to call. Running through all our functions made so far and recording timings can be done as follows:

```
import time
functions = [count_str0,
    count_str1,
    count_str2,
    count_str3,
    count_str4,
    count_str5]
timings = [] # timings[i] holds CPU time for functions[i]
for function in functions:
    t0 = time.process_time()
    function(dna, 'A')
    t1 = time.process_time()
    cpu_time = t1 - t0
    timings.append(cpu_time)
```

In Python, functions are ordinary objects so making a list of functions is no more special than making a list of strings or numbers.

We can now iterate over timings and functions simultaneously via zip to make a nice printout of the results:

```
for cpu_time, function in zip(timings, functions):
     print ('{f:<9s}: {cpu:.2f} s'.format(f=function.__name__, cpu=
        cpu_time))

count_str0: 0.92 s
count_str1: 0.85 s
count_str2: 1.49 s
count_str3: 2.25 s
count_str4: 2.13 s
count_str5: 0.05 s
```

It looks like count_str1 - the simple iteration over the string - is the best performer of all the user-defined functions. However, the built-in count functionality of strings (dna.count(base) in count_str5) runs almost 20 times faster than the best of the user defined Python functions! The reason is that the for loop needed to count in dna.count(base) is actually implemented in C and runs very much faster than loops in Python.

A clear lesson learned is: use the built-in functions if possible.

Chapter 7

File IO

Input data for a program often comes from files and the results of the computations are often written to files. We will discuss how Python can access information in files and how to create new files. We will focus mainly on text files, that is files that contain data readable by humans. Files that hold non-text data, also called binary files, are commonly used used over text files to pack data much more densely and provide much faster access. We will only consider binary files briefly in this course.

7.1 Reading from a text file

Remember the directory structure we created in week 1 (I have additional directories):

```
|____cds-230
| |____scripts
| |____data
| |____homework
| |____docs
| |____lectures
```

Suppose we have some measurements in a data file, call it **data1.txt**, stored in the **data** folder. Our goal is to read the the measurement values in **data1.txt**, find the average value, and print out the result.

Before trying to let a program read a file, we must know the file format, i.e., what the contents of the file look like, because the structure of the text in the file greatly influences the set of statements needed to read the file. We therefore start with viewing the contents of the file data1.txt. To this end, load the file into a text editor or viewer. What we see is a column with numbers:

```
21.8
18.1
19
23
26
17.8
```

Our task is to read this column of numbers into a list in the program and compute the average of the list items. To read from a text file, follow this basic paradigm:

- Open the file for reading.
- Read the line from the file.
- Process the line. In other words, do whatever you need to do with the line.
- Close the file.

To open a file for reading, call the built-in Python function **open**. It takes two parameters, both strings. The first parameter gives the name of the file. The second parameter should be the string ”r”, which indicates that you are reading from the file (if absent then ”r” is assumed). The function returns a reference to an *object* representing the file.

```
file_object = open(file_name, "r")
```

For example, to open the file data1.txt for reading - assuming the file is in the same directory where we are running the Python code:

```
file_name = "data1.txt"
in_file = open(file_name, "r")
```

When the location of a file is in another location, say in the **data** directory, then you will need to specify more that the file name, you will need to specify the **path** of the file. Depending on your operating system the paths may look different. For example:

1. Linux/Mac: ”/Users/jdoe/cds-230/data/data1.txt”
2. Linux/Mac: ”data/data1.txt”
3. Windows: ”C:\Users\jdoe\My Documents\cds-230\data\data1.txt”
4. Windows: ”\data\data1.txt”

An **absolute** filename gives a specific location on disk as in (1) or (3) above. A **relative** filename gives a location relative to the current working as in (2) or (4) above[1]. **Warning:** code will fail with a **FileNotFoundError** error if it fails to find the file specified in file_name.

So, assuming we are in the cds-230 directory:

```
file_name = "data/data1.txt"
in_file = open(file_name, "r")
```

Of course, you could get that down to a single line:

```
in_file = open("data/data1.txt", "r")
print(type(in_file))
<class '_io.TextIOWrapper'>
```

[1]A relative filename is usually a better choice

To read lines from the file, there are two general approaches:

Process entire file at once:

```python
all_data_in_one_big_string = in_file.read()
print(all_data_in_one_big_string)
21.8
18.1
19
23
26
17.8
```

This approach can be very useful when you are performing operations on the entire text. We will get back to this later.

Process one line at a time using a for-loop[2]

```python
for line in in_file:
    # Do something with line, e.g. print(line)
```

Here, in each iteration of the loop, the variable line will hold the current line from the file. *Notice that this type of for-loop differs from the for-loops we have already seen, in that what follows* in *is not a list, but a reference to a file object.*

In a file, each line ends with a newline character, which we indicate in Python by "\n". Each line that you read from the file will be a string containing all the characters in the line, including the newline character. Note that the newline character, "\n", is a **non-printing character**.

After the file is read, one should close the file object with close().

```python
in_file.close()
```

Instead of reading one line at a time, we can load all lines into a list of strings (lines) by (make sure you reopen the file):

```python
lines = in_file.readlines()
print(lines)
['21.8\n', '18.1\n', '19\n', '23\n', '26\n', '17.8\n']
```

Note that the output from readlines() allows you to "see" the "\n" characters. Using readlines() is equivalent to:

```python
lines = []
for line in in_file:
    lines.append(line)
```

or the list comprehension:

```python
lines = [line for line in in_file]
```

[2]Assumption: file is a **sequence** of lines, i.e. strings.

It is important to emphasize that **once you have read a file, the file pointer will be positioned at the end of the file** (also known as EOF). Therefore any subsequent read operation will yield nothing. In that case you will want to *rewind* the file. In that case you can use the **seek()** file function as follows:

```
# open a file
# read a file ...
#  ... I need to read the file again, so rewind
in_file.seek(0)
```

Here *seek*(0) is moving the file pointer back to the first position, i.e. 0, thus allowing you to read the file from the start. You can also use **seek()** to move to any position in the file.

Let's get back to our task. Once we have the data loaded into a we can compute the average of the numbers in the file. One way to do this is as follows:

```
mean = 0
for number in lines: # lines contain numbers
    mean = mean + float(number) # convert number to float
mean = mean/len(lines)
print(mean)
20.95
```

There is one important thing to note in the for loop: float(number). Python's reading functions return **text**, i.e. *str* objects. But since we need to perform numeric calculations we must convert the *str* objects, each line in the list, into numbers by using type casting.

Summing up a list of numbers is often done in numerical programs, so Python has a special function sum for performing this task. However, sum must in the present case operate on a list of floats, not strings. We can use a list comprehension to turn all elements in lines into corresponding float objects:

```
 mean = sum([float(line) for line in lines])/len(lines)
```

An alternative implementation is to load the lines into a list of float objects directly.

```
in_file = open('data/data1.txt')
numbers = [float(line) for line in in_file.readlines()]
in_file.close()
mean = sum(numbers)/len(numbers)
print(mean)
20.95
```

It is probably clear -or confusing - by now that one problem is solved by many alternative sets of statements, but this is the very nature of programming. Once we have gained experience we can judge several alternative solutions to a programming task and choose one that is either particularly compact, easy to understand, and/or easy to extend later.

7.2 Reading a Mixture of Text and Numbers

The data1.txt file has a very simple structure since it contains numbers only. Many data files contain a mix of text and numbers. The file data2.txt provides an example (see `http://www.`

worldclimate.com/cgi-bin/data.pl?ref=N41E012+2100+1623501G1):

```
Average rainfall (in mm) in Rome: 1188 months between 1782 and 1970
Jan 81.2
Feb 63.2
Mar 70.3
Apr  55.7
May  53.0
Jun  36.4
Jul  17.5
Aug  27.5
Sep  60.9
Oct  117.7
Nov  111.0
Dec  97.9
Year 792.9
```

How can we read the rainfall data in this file and extract the values?

The most straightforward solution is to read the file line by line, and for each line split the line into words, pick out the last (second) word on the line, convert this word to float, and store the float objects in a list. The complete code, wrapped in a function, may look like this

```python
def extract_data(filename):
    in_file = open(filename)
    in_file.readline() # skip the first line
    numbers = []
    for line in in_file:
        words = line.split()
        number = float(words[1])
        numbers.append(number)
    in_file.close()
    return numbers
values = extract_data('data2.txt')
print(values)
[63.2, 70.3, 55.7, 53.0, 36.4, 17.5, 27.5, 60.9, 117.7, 111.0, 97.9,
    792.9]
```

Note that the first line in the file is just a comment line and of no interest to us. We therefore read this line by in_file.readline(). The for loop over the lines in the file will then start from the next (second) line.

Can you think of a way to condense the program above?

7.3 With statements

Because things can go wrong when opening a file?for example, it may not exist or it may be corrupted?it is important to open files carefully so that errors may be handled gracefully. The *recommended* way to open a file in Python is to use a **with** statement. Its syntax is the following:

```
with expression as variable:
    # body
```

What happens when this statement executes is complicated, but when it is used to open a file, the steps are approximately like this:

1. An attempt is made to open the file.
2. If successful, the open file object is assigned to variable.
3. The body is executed.
4. The file is closed at the end, even if something goes wrong in the body.

The above example would be written as follows:

```
file_name = "data1.txt"
with open(file_name) as in_file:
    for line in in_file:
        print(line)
```

You can do all the usual file I/O operations that you would normally do as long as you are within the **with** code block. Once you leave that code block, the file handle will close and you will not be able to use it any more. You no longer have to close the file handle explicitly as the **with** operator does it automatically! See if you can change some of the earlier examples from this chapter so that they use the with method too.

7.4 Writing to a text file

Writing to a file is similar to reading from a file, but it differs in some important ways.

First, you have to open the file for writing. The second parameter to the open function should not be "r". Instead, you have your choice of "w" or "a". If you want to append to a file - that is, add to the end of an existing file - then choose "a". If you want to overwrite the file if it already exists, choose "w". If the file does not exist and you choose "a", you get the same effect as choosing "w". So, for example:

```
out_file = open(file_name, "w")
```

There is nothing special about the name out_file; use whatever name you like.

Just as you close a file that you are reading when you are done reading from it, you must close a file that you are writing when you are done writing to it.

To write to a file, call the **write** method on the file. This method takes one parameter, a string, and it writes to the end of the file. If you want a newline in the file, you have to put it into the string yourself.

Here is an example of writing a file:

```python
out_file = open("gettysburg.txt", "w")
out_file.write("Four score and seven years ago,\n")
out_file.write("our fathers brought forth on this continent, a new
    nation,\n")
out_file.write("conceived in Liberty, and dedicated to the
    proposition ")
out_file.write("that all men are created equal.\n")
out_file.close()
```

As another example consider writing a table to a file. Suppose we have an array of data

```python
data = [[ 0.75, 0.29619813, -0.29619813, -0.75 ],
    [ 0.29619813, 0.11697778, -0.11697778, -0.29619813],
    [-0.29619813, -0.11697778, 0.11697778, 0.29619813],
    [-0.75, -0.29619813, 0.29619813, 0.75 ]]
```

We iterate through the rows (first index) in the list, and for each row, we iterate through the column values (second index) and write each value to the file. At the end of each row, we must insert a newline character in the file to get a linebreak.

```python
out_file = open('my_table.dat', 'w')
for row in data:
    for column in row:
        out_file.write('{:14.8f}'.format(column))
    out_file.write('\n')
out_file.close()
```

The resulting data file becomes:

```
    0.75000000    0.29619813   -0.29619813   -0.75000000
    0.29619813    0.11697778   -0.11697778   -0.29619813
   -0.29619813   -0.11697778    0.11697778    0.29619813
   -0.75000000   -0.29619813    0.29619813    0.75000000
```

7.5 Numpy and Text IO

Numpy provides two functions to read in text data: savetxt and loadtxt. In the following simple example, we define an array x and save it as a textfile with savetxt:

```python
import numpy as np
x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]], np.int32)
np.savetxt("digits.dat", x)
```

The file ”digits.dat” is a text file and its content looks like this:

```
1.000000000000000000e+00 2.000000000000000000e+00 3.000000000000000000e+00
4.000000000000000000e+00 5.000000000000000000e+00 6.000000000000000000e+00
7.000000000000000000e+00 8.000000000000000000e+00 9.000000000000000000e+00
```

It is also possible to print the array in a special format, like for example with three decimal places or as integers. For this purpose we assign a format string to the third parameter 'fmt'. We saw in our first example that the default delimiter is a blank. We can change this behavior by assigning a string to the parameter "delimiter".

```
np.savetxt("digits2.dat", x, fmt="%2.3f", delimiter=",")
```

The file "digits2.dat" becomes:

```
1.000,2.000,3.000
4.000,5.000,6.000
7.000,8.000,9.000
```

We will read in now the file "digits.dat" using loadtxt:

```
y = np.loadtxt("digits.dat")
print(y)
[[1.  2.  3.]
 [4.  5.  6.]
 [7.  8.  9.]]
```

To read "digits2.dat":

```
y = np.loadtxt("digits2.dat", delimiter=",")
print(y)
[[1.  2.  3.]
 [4.  5.  6.]
 [7.  8.  9.]]

 print(type(y))
<class 'numpy.ndarray'>
```

Using the Numpy loadtxt reads data into numpy arrays which allows us to use the full Numpy functionality on the data.

7.6 Binary Files

Files that hold photographs, videos, zip files, executable programs, etc. are called binary files: they are not organized into lines, and cannot be opened with a normal text editor. However, Python works just as easily with binary files, but when we read from the file we are going to get bytes back rather than a string. Furthermore, you will find that the contents of a binary file will be, upon printing, unreadable.

In Python, the binary data is represented using a special type called **bytes**. The **bytes** type represents an immutable sequence of numbers between 0 and 255 [3].

To try to understand this, let's create a binary version of a simple text file and then try to read it. In Python reading and writing a binary file is done by appending b to the mode string:

[3] You may want to revisit section 9.3 now.

```python
with open('hello.bin', 'wb') as f:
    str = b'hello world!' # note the 'b' prefix
    f.write(str)

with open('hello.bin', 'rb') as f:
    print(f.read())
b'hello world!'  # note the 'b' prefix
```

Note that the input to write() must be of **bytes** type, i.e. to create a bytes-like object from a string we prefix the string with a *b*. Similarly, the output from read is in **bytes** and so Python prefixes a *b* to the string. The *b* means that those are byte objects. Since bytes are just 1s and 0s we need to to **decode** or **encode** these objects whenever we read or write in binary mode.

Encoding[4] and decoding[5] refer to an intermediate representation of the text process so that we can work with data on a computer. After all data are internally represented as 1s and 0s but what we get when we read a file is numbers and text. There are several encodings and Python uses one called **UTF-8**. For the hello.bin data we do the following:

```python
with open('hello.bin', 'rb') as f:
    data = f.read()
    text = data.decode('utf-8') # from binary to text
    print(text)
hello world!   # note that there is no 'b'

with open('hello.bin', 'wb') as f:
    text = 'hello world!'
    f.write(text.encode('utf-8')) #from text to binary
```

The encode and decode functions accept a few parameters. The first and most important one, is the parameter that will indicate the encoding system. As shown above we used 'utf-8' to decode data from binary to text and also 'utf-8' to encode from text to binary.

Finally, it is important to note that, in our case, binary data happens to contain printable characters, like alphabets, newline etc. However, this will not be the case most of the time. It means that with binary data we cannot reliably use readline() and file object (as an iterator) to read the contents of a file because might be no newline character in a file. The best way to read binary data is to **read it in chunks** using the read() method.

7.7 More on encoding and decoding: Unicode characters

If you find yourself dealing with text that contains non-ASCII[6] characters, you have to learn about **Unicode** - what it is, how it works, and how Python uses it.

Learning about Unicode is beyond the scope of this class. However, we can try to get a general idea as to how Python deals with it. First, you must understand the difference between bytes and characters. In older, ASCII-centric languages and environments, bytes and characters are treated

[4]The process of turning abstract symbols into bits

[5]The process of reading bits (1s and 0s), making sense out of them, and getting back symbols or characters

[6]Remember that the original ASCII character set encodes128 specified characters into seven-bit integers and can be used to represent all written text used in the English language.

as the same thing. Since a byte can hold up to 256 values, these environments are limited to 256 characters. Unicode, on the other hand, has tens of thousands of characters. That means that each Unicode character takes more than one byte, so you need to make the distinction between characters and bytes.

Standard Python strings are really byte strings, and a Python character is really a byte. Other terms for the standard Python type are "8-bit string" and "plain string."

Conversely, a Python Unicode character is an abstract object big enough to hold the character, analogous to Python's long integers. You don't have to worry about the internal representation; the representation of Unicode characters becomes an issue only when you are trying to send them to some byte-oriented function, such as the **write** method for files. At that point, you must choose how to represent the characters as bytes. Converting from Unicode to a byte string is called **encoding** the string. Similarly, when you load Unicode strings from a file or other byte-oriented object, you need to **decode** the strings from bytes to characters.

There are many ways of converting Unicode objects to byte strings, each of which is called an encoding. For a variety of historical, political, and technical reasons, there is no one "right" encoding. Every encoding has a case-insensitive name, and that name is passed to the decode method as a parameter. Here are a few you should know about:

- The **UTF-8** encoding can handle any Unicode character. It is also backward compatible with ASCII, so a pure ASCII file can also be considered a UTF-8 file, and a UTF-8 file that happens to use only ASCII characters is identical to an ASCII file with the same characters. This property makes UTF-8 very backward-compatible, especially with older Unix tools. UTF-8 is far and away the dominant encoding on Unix. Its primary weakness is that it is fairly inefficient for Eastern texts.

- The **UTF-16** encoding is favored by Microsoft operating systems and the Java environment. It is less efficient for Western languages but more efficient for Eastern ones. A variant of UTF-16 is sometimes known as UCS-2.

- The **ISO-8859** series of encodings are 256-character ASCII supersets. They cannot support all of the Unicode characters; they can support only some particular language or family of languages. ISO-8859-1, also known as Latin-1, covers most Western European and African languages, but not Arabic. ISO-8859-2, also known as Latin-2, covers many Eastern European languages such as Hungarian and Polish.

If you want to be able to encode all Unicode characters, you probably want to use UTF-8. You will probably need to deal with the other encodings only when you are handed data in those encodings created by some other application.

7.7.1 ord() and chr()

Given a string representing one Unicode character, $ord()$ returns an integer representing the Unicode code point of that character. For example, $ord('a')$ returns the integer 97. Conversely, $chr(i)$ returns the string representing a character whose Unicode code point is the integer i. For example, $chr(97)$ returns the string $'a'$:

```
ord('a')
```

```
>>> 97
ord('7')
>>> 55
ord('%')
>>> 37

# Conversely:
chr(97)
'a'
chr(55)
'7'
chr(37)
'%'
```

Chapter 8

Python Arrays

8.1 Lists as Arrays

A data structure is a way to organize data that we wish to process with a computer program. A one-dimensional array (or array) is a data structure that stores a sequence of (references to) objects. We refer to the objects within an array as its elements. The method that we use to refer to elements in an array is numbering and then indexing them. If we have n elements in the sequence, we think of them as being numbered from 0 to n - 1. Then, we can unambiguously specify one of them by referring to the i^{th} element for any integer i in this range.

```
Arrays consist of fixed-size data records that allow each element to be
efficiently located based on its findex.
```

A two-dimensional array is an array of (references to) one-dimensional arrays. Whereas the elements of a one-dimensional array are indexed by a single integer, the elements of a two-dimensional array are indexed by a pair of integers: the first specifying a row, and the second specifying a column. In this chapter we will work with **lists as arrays** and refer to them as arrays thus setting the tone for next chapter. We will use arrays to perform various computations. The chapter will also serve as a review of most of the concepts learned so far and in fact the array descriptions are identical to that of lists first introduced in chapter 4.1.

Initializing Python arrays. The simplest way to create an array in Python is to place comma-separated literals between matching square brackets. For example, the code

```python
SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
x = [0.30, 0.60, 0.10]
y = [0.50, 0.10, 0.40]
```

creates an array SUITS[] with four strings, and creates arrays x[] and y[], each with three floats.

It is useful to think of references of the elements in an array as stored contiguously, one after the other, in your computer's memory, as shown in figure 8.1 for the SUITS[] array defined above.

Zero-based indexing. We always refer to the first element of an array a[] as a[0], the second as a[1], and so forth. It might seem more natural to refer to the first element as a[1], the second

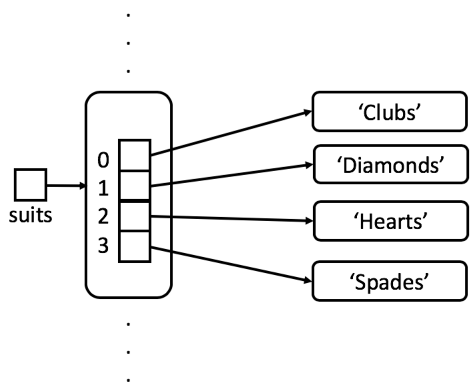


Figure 8.1: Array data structure

element as a[2], and so forth, but starting the indexing with 0 has some advantages and has emerged as the convention used in most modern programming languages.

Array length. You can access the length of an array using Python's built-in len() function: len(a) is the number of elements in a[]. In Python, we can use the += operator to append elements to an array. For example, if a[] is the array [1, 2, 3], then the statement a += [4] extends it to [1, 2, 3, 4]. More generally, we can make an array of n floats, with each element initialized to 0.0, with the code:

```
a = []
for i in range(n):
    a += [0.0]
```

Bounds checking. You must be careful when programming with arrays. It is your responsibility to use legal indices when accessing an array element.

Mutability. An object is mutable if its value can change. Arrays are mutable objects because we can change their elements. For example, if we create an array x = [.30, .60, .10], then the assignment statement x[1] = .99 changes it to the array [.30, .99, .10]. The following code **reverses** the order of the elements in an array a[]:

```
n = len(a)
for i in range(n // 2):
    temp = a[i]
    a[i] = a[n-1-i]
    a[n-1-i] = temp
```

Of course, since a is a list, you could simply use a.reverse() which modifies the original array in-place , i.e. no additional memory is required.

Iteration. The following code iterates over all elements of an array to compute the average of the floats that it contains:

```python
total = 0.0
for i in range(len(a)):
    total += a[i]
average = total / len(a)
```

Python also supports iterating over the elements in an array without referring to the indices explicitly. To do so, put the array name after the in keyword in a for statement, as follows:

```python
total = 0.0
for v in a:
    total += v
average = total / len(a)
```

Writing an array. You can write an array by passing it as an argument to print(). Each object in the array is converted to a string.

Aliasing. If x[] and y[] are arrays, the statement x = y causes x and y to reference the same array. This result has an effect that is perhaps unexpected, at first, because it is natural to think of x and y as references to two independent arrays. For example, after the assignment statements

```python
x = [.30, .60, .10]
y = x
x[1] = .99
```

y[1] is also .99, even though the code does not refer directly to y[1]. This situation whereby two variables refer to the same object is known as **aliasing**.

Copying and slicing. So how do we make a copy y[] of a given array x[]? One answer to this question is to iterate through x[] to build y[], as in the following code:

```python
y = []
for v in x:
    y += [v]
```

Copying an array is such a useful operation that Python provides language support for a more general operation known as **slicing**. So, the expression a[i:j] evaluates to a new array whose elements are a[i], ..., a[j-1]. Moreover, the default value for i is 0 and the default value for j is len(a), so y = x[:] is equivalent to the code given above.

Example: Dot Product. Given two vectors of the same length, their **dot product** is the sum of the products of their corresponding components. If we represent the two vectors as one-dimensional arrays x[] and y[] that are each of length n, their dot product is easy to compute:

```python
dotp = 0.0
for i in range(n):
    dotp += x[i]*y[i]
```

Example: Representing playing cards. Suppose that we want to compose a program that processes playing cards. We might start with the following code:

```python
SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
RANKS = ['2', '3', '4', '5', '6', '7', '8', '9', '10',
         'Jack', 'Queen', 'King', 'Ace']
```

For example, we might use these two arrays to write a random card name, as follows:

```python
rank = random.randrange(0, len(RANKS))
suit = random.randrange(0, len(SUITS))
print((RANKS[rank] + ' of ' + SUITS[suit]))
```

We might use the following code to initialize an array of length 52 that represents a deck of playing cards, using the two arrays just defined:

```python
deck = []
for rank in RANKS:
    for suit in SUITS:
        card = rank + ' of ' + suit
        deck += [card]
print(deck)
```

Frequently, we wish to **exchange** two elements in an array. Continuing our example with playing cards, the following code exchanges the cards at indices i and j:

```python
temp = deck[i]
deck[i] = deck[j]
deck[j] = temp
```

Finally, the following code **shuffles** our deck of cards:

```python
n = len(deck)
for i in range(n):
    r = random.randrange(i, n)
    temp = deck[r]
    deck[r] = deck[i]
    deck[i] = temp
print(deck)
```

8.2 2D Arrays

In many applications, a convenient way to store information is to use a table of numbers organized in a rectangular table and refer to rows and columns in the table. The mathematical abstraction corresponding to such tables is a **matrix**; the corresponding data structure is a **two-dimensional array**.

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ . & . & . & . & . \\ . & . & . & . & . \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}$$

Here, the numbers are arranged in rows and columns, where m and n represent the total number of rows and columns respectively. So, an example of a *square* 2x2 matrix is

$$A = \begin{bmatrix} 1 & 5 \\ -9 & 2 \end{bmatrix}$$

while a 3x4 matrix is

$$B = \begin{bmatrix} 4 & -3 & 11 & -13 \\ 1 & 0 & 7 & 20 \\ -12 & 2 & 5 & 6 \end{bmatrix}$$

Traditionally, we use the letter i to represent the row number and the letter j to represent the column number. So, if $i = 2$ and $j = 3$ then $B(i, j) = 7$. In Python, however, we need to keep in mind that we use zero-based indexing.

Initialization. The simplest way to create a two-dimensional array is to place comma-separated one-dimensional arrays between matching square brackets. For example, this matrix of integers having two rows and three columns

```
18 19 20
21 22 23
```

could be represented in Python using this array of arrays:

```python
a = [[18, 19, 20], [21, 22, 23]]
```

Notice the following:

```python
print(len(a)) # 2 : The array a is basically a "list" with two lists
print(len(a[0])) # 3 :  The first row is a "list" has 3 elements
print(len(a[1])) # 3 :  The second row has 3 elements
```

More generally, Python represents an m-by-n array as an array that contains m objects, each of which is an array that contains n objects. For example, this Python code creates an m-by-n array a[][] of floats, with all elements initialized to 0.0:

```python
m=2
n=3
a = []
for i in range(m):
    row = [0.0] * n
    a += [row]
```

For convenience we can make this into a function that we can use later on:

```python
def initMat(m, n):
    mat = []
    for i in range(m):
        row = [0.0] * n
        mat += [row]
    return mat
```

Indexing. When a[][] is a two-dimensional array, the syntax a[i] denotes a reference to its i^{th} row. The syntax a[i][j] refers to the object at row i and column j. To access each of the elements in a two-dimensional array, we use two nested for loops. For example, this code writes each object of the 3-by-4 array B given above, one row per line.

```python
B = [[4, -3, 11, -13], [1, 0, 7, 20], [-12, 2, 5, 6]]
for i in range(len(B)):
    for j in range(len(B[0])):
        print(B[i][j], end=' ')
    print()

# will print:
4 -3 11 -13
1 0 7 20
-12 2 5 6
```

Matrix operations. Typical applications in science and engineering involve representing matrices as two-dimensional arrays and then implementing various mathematical operations with matrix operands. For example, we can **add** two matrices a[][] and b[][] as follows:
par

Adding matrices.You can only add and subtract matrices of the same dimensions (size and shape), which means that you can add or subtract only the *corresponding elements*. Here is an example of how to add two 2x2 matrices:

```python
def addMatrices(a,b):
    '''adds two 2x2 matrices together'''
    c = [[a[0][0]+b[0][0],a[0][1]+b[0][1]],
    [a[1][0]+b[1][0],a[1][1]+b[1][1]]]
    return c # note c is an array!

a = [[2,3],[5,-8]]
b = [[1,-4],[8,-6]]
c = addMatrices(a,b)
print(c)

[[3, -1], [13, -14]]
```

An even better way is to use loops and then we can generalize to add 2 nxn matrices:

```python
def addMatrices(a,b):
    '''adds two nxn matrices together'''
    c = initMat(len(a), len(a[0])) # note this initialization
    for i in range(len(a)): # loop over rows
        for j in range(len(a[0])): # loop over columns
            c[i][j] = a[i][j] + b[i][j]
    return c

a = [[2,3],[5,-8]]
b = [[1,-4],[8,-6]]
print(addMatrices(a,b))

[[3, -1], [13, -14]]
```

Note the use if initMat().

Multiplying matrices. Similarly, we can **multiply** two matrices $\vec{a}\vec{b}$ keeping in mind that $\vec{a}\vec{b}$ is defined if and only if the number of columns of $\vec{a}$ equals the number of rows of $\vec{b}$. Thus, each element c[i][j] in the product of a[][] and b[][] is computed by taking the dot product of row i of a[][] with column j of b[][]. For example, a general formula to multiply two 2x2 matrices is:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & bf + bh \\ ce + dg & ch + dh \end{bmatrix} \tag{8.1}$$

This can be generalized to multiplication of (nxm)×(mxp). This may seem complicated but once we have the matrices as input we can figure out the number of rows and columns and implement an algorithm to carry out the multiplication.

We can implement matrix multiplication as follows:

```
def multMatrices(a ,b):
    c = initMat(len(a), len(b[0])) # initialize 2D matrix
    for i in range(len(a)): # iterate over 2 rows of a
        for j in range(len(b[0])): # iterate over 2 columns of b
            for k in range(len(b)): # iterate over rows/columns of b
                /a
                c[i][j] += a[i][k] * b[k][j] # dot product
    return c

a = [[1, 2, 3], [4, 5, 6]] # 2x3 matrix
b = [[1, 2], [3, 4], [5, 6]]  # 3x2 matrix
for r in multMatrices(a ,b):
    print(r)

[22, 28]
[49, 64]
```

Although this is a straightforward implementation of matrix multiplication you should understand how it works. This implementation is simple but inefficient, especially for large arrays like the ones used in science and engineering. We will introduce more efficient ways to deal with numerical array computations in the next chapter.

Order Matters in Matrix Multiplication. An important fact about multiplying matrices is that AxB doesn't necessarily equal BxA, i.e. matrix multiplication is non-commutative. Try it with the example above.

Transforming Matrices. Multiplying matrices lets us transform them. For example, given a vector v one can *rotate* v by multiplying it with a rotation matrix R and obtain a new, rotated, vector v':

$$v' = Rv \tag{8.2}$$

where R is the rotation matrix[1] in 2D:

$$R = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

Note that for the given R the dimensions of v (and v') must be 2x1. Now, if $\theta = 90^o$ then R is going to rotate v in the **counterclockwise** direction and R will be given by

$$R = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

For example, if we start with a unit vector lying on the x-axis and rotate it by 90^o then we'll end up with a unit vector lying along the y-axis:

$$\begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix}$$

And in Python this becomes:

```python
R = [[0, -1],[1, 0]] # 90deg counter-clockwise Rotation matrix
v = [[1],[0]] # 2x1 vector, lying along x-axis
vp = multMatrices(R ,v) # rotate 90 degrees
print(vp)

[[0.0], [1.0]]  # rotated 2x1 vector, now lying along y-axis
```

Of course, we can also rotate another matrix, say A, but that is a bit more difficult to visualize than with vectors.

Transposing matrices. An important concept in matrices is **transposition**, where the columns become the rows, and vice versa. In our example, we want to change some matrix B into B^T, the notation for "the B-matrix, transposed.". So, if

$$B = \begin{bmatrix} 4 & -3 & 11 & -13 \\ 1 & 0 & 7 & 20 \\ -12 & 2 & 5 & 6 \end{bmatrix}$$

then

$$B^T = \begin{bmatrix} 4 & 1 & -12 \\ -3 & 0 & 2 \\ 11 & 7 & 5 \\ -13 & 20 & 6 \end{bmatrix}$$

[1] `https://www.wikiwand.com/en/Rotation_matrix`

We will be returning to all the matrix operations and concepts discussed in this chapter. However, we will be using a powerful Python package called NumPy.

8.3 Other Arrays in Python

This section is optional.

8.3.1 Basic Typed Arrays

Python's **array** module provides space-efficient storage of basic C-style data types like bytes, 32-bit integers, floating point numbers, and so on. We will not be using them in this class but it is worth mentioning them.

Arrays created with the **array** module are mutable and behave similarly to lists - except they are "typed arrays" constrained to a single data type.

Because of this constraint **array** objects with many elements are more space-efficient than lists (and tuples). The elements stored in them are tightly packed and this can be useful if you need to store many elements of the same type.

Also, **array** arrays support many of the same methods as regular lists. For example, to append to an array in Python you can just use the familiar array.append() method.

As a result of this similarity between Python lists and array objects, you might be able to use it as a "drop-in replacement" without requiring major changes to your application.

```python
import array
arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
arr[1]
1.5

# Arrays are "typed":
arr[1] = 'hello'
TypeError: "must be real number, not str"
```

8.3.2 Immutable Arrays of Unicode Characters

Python 3.x uses **str** objects to store textual data as immutable sequences of **Unicode** characters. Practically speaking that means a **str** is an immutable array of characters. Oddly enough it's also a recursive data structure - each character in a string is a **str** object of length 1 itself.

String objects are space-efficient because they are tightly packed and specialize in a single data type. If you are storing Unicode text you should use them. Because strings are immutable in Python modifying a string requires creating a modified copy. The closest equivalent to a "mutable string" is storing individual characters inside a list.

```python
arr = 'abcd'
arr[1]
'b'

```

```
arr
'abcd'

# Strings are immutable:
arr[1] = 'e'
TypeError: "'str' object does not support item assignment"

# Strings are recursive data structures:
type('abc')
"<class 'str'>"
type('abc'[0])
"<class 'str'>"
```

8.3.3 Immutable Arrays of Single Bytes

Bytes objects are immutable sequences of single bytes (integers in the range of $0 \leq x \leq 255$). Conceptually they are similar to **str** objects and you can also think of them as immutable arrays of bytes.

Like strings, bytes have their own literal syntax for creating objects and they are space-efficient. Bytes objects are immutable, but unlike strings there is a dedicated ”mutable byte array” data type called **bytearray** that they can be unpacked into.

```
arr = bytes((0, 1, 2, 3))
arr[1]
1

# Bytes literals have their own syntax:
arr
b'\x00\x01\x02\x03'

# Only valid "bytes" are allowed:
bytes((0, 300))
ValueError: "bytes must be in range(0, 256)"

# Bytes are immutable:
arr[1] = 23
TypeError: "'bytes' object does not support item assignment"
```

8.3.4 Mutable Arrays of Single Bytes

The **bytearray** type is a mutable sequence of integers in the range $0 \leq x \leq 255$. They are closely related to bytes objects with the main difference being that **bytearray**s can be modified freely - you can overwrite elements, remove existing elements, or add new ones. The **bytearray** object will grow and shrink appropriately.

```
arr = bytearray((0, 1, 2, 3))
arr[1]
```

```
1

# Bytearrays are mutable:
arr[1] = 23
arr
bytearray(b'\x00\x17\x02\x03')

arr[1]
23
```

There are a number of built-in data structures you can choose from when it comes to implementing arrays in Python. For simple applications, that may be all that you need. However, for scientific computing the most popular choice is use the NumPy module which we will cover next.

Chapter 9

Numpy

An array is an indexed sequence of objects, all of which are of the same type. Earlier, we implemented arrays using the Python list data type: a list object is an indexed sequence of objects, not necessarily of the same type. Using Python lists to implement arrays incurs substantial overhead, both in terms of memory (because Python must associate type information with each element) and time (because Python must perform a type check when accessing an element). Moreover, it is the programmer's responsibility to enforce the "all elements of the same type" constraint.

Now, we describe an alternative way to represent arrays in Python using the ndarray ("n-dimensional array") data type in the standard NumPy library: a ndarray object is an indexed sequence of objects, all of which are of the the same type - and NumPy enforces the "all elements of the same type" constraint. We use the informal term NumPy array to mean "an object of type ndarray."

Typically the elements of a NumPy array are numbers, such as floats or integers. As a result, there is minimal overhead in terms of memory (because NumPy need only associate type information with the array and not each element). Also, this representation can dramatically speed up certain types of "vectorized" computations because the array elements are stored contiguously in memory.

But there is more to NumPy than numeric arrays: the NumPy library also supports arrays whose elements are booleans and strings, and arrays whose elements are of data types that you define. There also is more to NumPy than arrays: the NumPy library provides many functions that work on "scalar" numeric objects. But the most valuable aspect of NumPy is its ability to manipulate numeric arrays; this chapter describes only that aspect.

9.1 When to Use Numpy

When should you use NumPy? One short answer is "when you need the functionality that NumPy provides." Indeed, as described later, NumPy provides a rich set of numeric array-handling functions and methods. Generally you should use the pre-defined (and thoroughly tested) functions and methods provided by NumPy instead of defining your own equivalent functions or methods.

Another short answer is "when you need your program to run faster." **The NumPy library was not written in Python**; instead it was written using the C programming language. Programs written in C run more quickly than those written in Python. So a Python program that implements arrays as NumPy arrays (maybe) will run faster than an equivalent Python program that implements arrays as ordinary Python lists.

However, there is more to the story. Since NumPy was written in C, there is a boundary between NumPy code and ordinary Python code. Crossing that boundary is expensive. That is, calling a NumPy function or method from ordinary Python code consumes more time than does calling an ordinary Python function or method. Similarly, returning a value from a NumPy function or method to ordinary Python code consumes more time than does returning a value from an ordinary Python function or method. With that in mind, suppose:

- Program A calls NumPy functions/methods many times, and each function/method call involves little computation.
- Program B calls NumPy functions/methods few times, and each function/method call involves much computation.

In that case Program B probably will benefit from the use of NumPy, but Program A probably will not.

9.2 Numpy Data Types

Since the NumPy library was written in the C programming language, its fundamental data types are, for the most part, those of C. Table 9.1 (taken from the online NumPy documentation) lists the NumPy fundamental data types: When you create a NumPy array, you specify the type of the array's elements. Normally you specify the element data type as a Python data type: int, float, bool, or complex. The Python int data type maps to the NumPy int_ data type. So if you create a NumPy array with elements of data type int, then internally within NumPy its elements are of type int_. Similarly, the Python float data type maps to the NumPy float_ data type, the Python bool data type maps to the NumPy bool_ data type, and the Python complex data type maps to the NumPy complex_ data type.

Usually you need not be concerned about the distinction between Python data types and NumPy data types. When an object of a Python data type is sent ”across the boundary” to NumPy, the object automatically is converted to the appropriate Numpy data type. Conversely, when NumPy sends an object of a NumPy data type back to Python, and when that object is used in a context that demands an object of a Python data type, the object automatically is converted to the appropriate Python data type.We will henceforth ignore the distinction between Python and NumPy data types.

However the distinction between Python and NumPy data types can be important in programs that manipulate large-magnitude integers, that is, integers whose absolute values are large. Whereas Python int objects have unlimited range, the range of NumPy int_ objects is limited. A NumPy int_ object has range -2147483648 to 2147483647 (that is -2^{31} to $2^{31} - 1$) on systems that store integers using 32 binary digits, and -9223372036854775808 to 9223372036854775807 (that is, -2^{63} to $2^{63} - 1$) on systems that store integers using 64 binary digits.

So in NumPy it is possible for an expression to evaluate to an integer that is outside of the range that NumPy can store. When such an *overflow* occurs, NumPy evaluates the expression to an integer that is mathematically incorrect. Beware when manipulating large-magnitude integers in NumPy.

Table 9.1: NumPy Data Types

Data	Type Description
bool_	boolean (True or False) stored using 8 bits
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float_	Shorthand for float64
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

9.3 NumPy Array Fundamentals

There are several ways to import NumPy. The standard approach is to use a simple import statement:

```
import numpy
```

However, for large amounts of calls to NumPy functions, it can become tedious to write numpy.X over and over again. Instead, it is common to import under the briefer name np:

```
import numpy as np
```

This statement will allow us to access NumPy objects using np.X instead of numpy.X. It is also possible to import NumPy directly into the current namespace so that we don't have to use dot notation at all, but rather simply call the functions as if they were built-in:

```
from numpy import *
```

Never do this. If you do you will remove some of the nice organization that modules provide as well as pontentially create namespace collisions. In this class we will assume that **import numpy as np** has been used.

How do we create Numpy arrays? An array can be created from a list:

```python
import numpy as np
a = np.array([18, 19, 20, 21], int)
```

creates a one-dimensional NumPy array containing integers 18, 19, 20, and 21, and this statement:

```python
b = np.array([[18.5, 19.3], [20.1, 21.0],  [23.7, 24.9]], float)
```

creates a two dimensional NumPy array of floats having three rows and two columns. If you omit the second argument to np.array(), then the function infers the desired element type by examining the types of the values provided in the first argument.

To convert a NumPy array to a Python list, call the tolist() method. For example the expression a.tolist() evaluates to [18, 19, 20, 21], and b.tolist() evaluates to [[18.5, 19.3], [20.1, 21.0], [23.7, 24.9]].

You can reference an element of a one-dimensional NumPy array, just as you can reference an elements of a Python list, by specifying an index enclosed within square brackets. For example a[1] evaluates to 19. To reference an element of a two-dimensional NumPy array, specify the indices within square brackets, separated by commas. For example b[1, 0] evaluates to 20.1. Note that the syntax for referencing an element of a NumPy two-dimensional array differs from the syntax for referencing an element of a list of lists. (Recall that you would use the expression b[1][0] if b referenced a Python list of lists.)

Iteration over NumPy arrays works as expected:

```python
for element in a:
    print(element)

for row in b:
    for element in row:
        print(element)
```

Slicing a NumPy array also is straightforward. For example a[1:2] evaluates to the NumPy array [19, 20]. However, *slicing a numpy array does not generate a copy of the array.* For example, this statement:

```python
e = a[1:2]
```

causes e to reference a subarray of the array referenced by a that is not distinct from a. Assigning some value to e[0] would change both e[0] and a[1]. Accordingly, the expression a[:] does not create a copy of the NumPy array referenced by a. Instead, to make a copy of a NumPy array you must call the copy() method:

```python
e = a.copy()
```

Arrays can be multidimensional. Unlike lists, different axes are accessed using commas inside bracket notation. Here is an example with a two-dimensional array (e.g., a matrix):

```python
a = np.array([[1, 2, 3], [4, 5, 6]], float)
print(a)
array([[1., 2., 3.],
       [4., 5., 6.]])
```

Array slicing works with multiple dimensions in the same way as usual, applying each slice specification as a filter to a specified dimension. Use of a single ":" in a dimension indicates the use of everything along that dimension:

```
a = np.array([[1, 2, 3], [4, 5, 6]], float)
a[1,:]
array([ 4., 5., 6.])

a[:,2]
array([ 3., 6.])

a[-1:,-2:]
array([[ 5., 6.]])
```

Array Properties The *shape* property of an array returns a tuple with the size of each array dimension:

```
a.shap(2,3)
```

The *dtype* property tells you what type of values are stored by the array:

```
a.dtype
dtype('float64')
```

When used with an array, the *len* function returns the length of the first axis:

```
len(a)
2
```

The *in* statement can be used to test if values are present in an array:

```
2 in a
True
0 in a
False
```

Arrays can be reshaped using tuples that specify new dimensions. In the following example, we turn a ten-element one-dimensional array into a two-dimensional one whose first axis has five elements and whose second axis has two elements:

```
a = np.array(range(10), float)
a
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
a = a.reshape((5, 2))
a
array([[ 0., 1.],
       [ 2., 3.],
       [ 4., 5.],
       [ 6., 7.],
       [ 8., 9.]])
a.shape
(5, 2)
```

Notice that the reshape function creates a new array and does not itself modify the original array.

Transposed versions of arrays can also be generated, which will create a new array with the final two axes switched:

```
a = np.array(range(6), float).reshape((2, 3))
a
array([[ 0., 1., 2.],
       [ 3., 4., 5.]])
a.transpose()
array([[ 0., 3.],
       [ 1., 4.],
       [ 2., 5.]])
```

One-dimensional versions of multi-dimensional arrays can be generated with flatten:

```
a = np.array([[1, 2, 3], [4, 5, 6]], float)
a
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])
a.flatten()
array([ 1., 2., 3., 4., 5., 6.])
```

Finally, the dimensionality of an array can be increased using the newaxis constant in bracket notation:

```
a = np.array([1, 2, 3], float)
a
array([1., 2., 3.])
a[:,np.newaxis]    # Note that this is now a column vector
array([[ 1.],
       [ 2.],
       [ 3.]])
a[:,np.newaxis].shape
(3,1)
b[np.newaxis,:]
array([[ 1., 2., 3.]])
b[np.newaxis,:].shape
(1,3)
```

Notice here that in each case the new array has two dimensions; the one created by newaxis has a length of one. The newaxis approach is convenient for generating the proper dimensioned arrays for vector and matrix mathematics.

9.3.1 Other ways to create arrays

The arange function is similar to the range function but returns an array:

```
np.arange(5, dtype=float)
array([ 0., 1., 2., 3., 4.])
np.arange(1, 6, 2, dtype=int)
```

```
array([1, 3, 5])
```

The functions zeros and ones create new arrays of specified dimensions filled with these values. These are perhaps the most commonly used functions to create new arrays:

```
np.ones((2,3), dtype=float)
array([[ 1., 1., 1.],
 [ 1., 1., 1.]])
np.zeros(7, dtype=int)
array([0, 0, 0, 0, 0, 0, 0])
```

There are also a number of functions for creating special matrices (2D arrays). To create an identity matrix of a given size,

```
np.identity(4, dtype=float)
array([[ 1., 0., 0., 0.],
       [ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 0., 0., 1.]])
```

The eye function returns matrices with ones along the kth diagonal:

```
np.eye(4, k=1, dtype=float)
array([[ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 0., 0., 1.],
       [ 0., 0., 0., 0.]])
```

9.4 NumPy Array Operations

9.4.1 Basic operations

Many functions exist for extracting whole-array properties. The items in an array can be summed or multiplied:

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0
```

In this example, member functions of the arrays were used. Alternatively, standalone functions in the NumPy module can be accessed:

```
>>> np.sum(a)
9.0
>>> np.prod(a)
24.0
```

For most of the routines described below, both standalone and member functions are available. A number of routines enable computation of statistical quantities in array datasets, such as the mean (average), variance, and standard deviation:

```
>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>> a.var()
12.666666666666666
>>> a.std()
3.5590260840104371
```

It's also possible to find the minimum and maximum element values:

```
>>> a = np.array([2, 1, 9], float)
>>> a.min()
1.0
>>> a.max()
9.0
```

The argmin and argmax functions return the array indices of the minimum and maximum values:

```
>>> a = np.array([2, 1, 9], float)
>>> a.argmin()
1
>>> a.argmax()
2
```

For multidimensional arrays, each of the functions thus far described can take an optional argument axis that will perform an operation along only the specified axis, placing the results in a return array:

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2., 2.])
>>> a.mean(axis=1)
array([ 1., 1., 4.])
>>> a.min(axis=1)
array([ 0., -1., 3.])
>>> a.max(axis=0)
array([ 3., 5.])
```

Like lists, arrays can be sorted:

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> sorted(a)
[-1.0, 0.0, 2.0, 5.0, 6.0]
>>> a.sort()
>>> a
array([-1., 0., 2., 5., 6.])
```

For multidimensional arrays, each of the functions thus far described can take an optional argument axis that will perform an operation along only the specified axis, placing the results in a return array:

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2., 2.])
>>> a.mean(axis=1)
array([ 1., 1., 4.])
>>> a.min(axis=1)
array([ 0., -1., 3.])
>>> a.max(axis=0)
array([ 3., 5.])
Like lists, arrays can be sorted:
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> sorted(a)
[-1.0, 0.0, 2.0, 5.0, 6.0]
>>> a.sort()
>>> a
array([-1., 0., 2., 5., 6.])
```

9.4.2 Comparison operators and value testing

Boolean comparisons can be used to compare members elementwise on arrays of equal size. The return value is an array of Boolean True / False values:

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
```

The results of a Boolean comparison can be stored in an array:

```
>>> c = a > b
>>> c
array([ True, False, False], dtype=bool)
```

The any and all operators can be used to determine whether or not any or all elements of a Boolean array are true:

```
>>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

9.4.3 Array item selection and manipulation

We have already seen that, like lists, individual elements and slices of arrays can be selected using bracket notation. Unlike lists, however, arrays also permit selection using other arrays. That is, we can use array selectors to filter for specific subsets of elements of other arrays. Boolean arrays can be used as array selectors:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> a >= 6
array([[ True, False],
 [False, True]], dtype=bool)
>>> a[a >= 6]
array([ 6., 9.])
```

Notice that sending the Boolean array given by a¿=6 to the bracket selection for a, an array with only the True elements is returned. We could have also stored the selector array in a variable:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> sel = (a >= 6)
>>> a[sel]
array([ 6., 9.])
```

In addition to Boolean selection, it is possible to select using integer arrays. Here, the integer arrays contain the indices of the elements to be taken from an array. Consider the following one-dimensional example:

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a[b]
array([ 2., 2., 4., 8., 6., 4.])
```

9.4.4 Vector and matrix mathematics

NumPy provides many functions for performing standard vector and matrix multiplication routines. To perform a dot product,

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
```

The dot function also generalizes to matrix multiplication:

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
>>> a
array([[ 0., 1.],
 [ 2., 3.]])
>>> np.dot(b, a)
array([ 6., 11.])
>>> np.dot(a, b)
array([ 3., 13.])
>>> np.dot(a, c)
```

```
array([[ 4., 0.],
 [ 14., 2.]])
>>> np.dot(c, a)
array([[ 2., 4.],
 [ 0., 4.]])
```

It is also possible to generate inner, outer, and cross products of matrices and vectors. For vectors, note that the inner product is equivalent to the dot product:

```
>>> a = np.array([1, 4, 0], float)
>>> b = np.array([2, 2, 1], float)
>>> np.outer(a, b)
array([[ 2., 2., 1.],
 [ 8., 8., 4.],
 [ 0., 0., 0.]])
>>> np.inner(a, b)
10.0
>>> np.cross(a, b)
array([ 4., -1., -6.])
```

NumPy also comes with a number of built-in routines for linear algebra calculations. These can be found in the sub-module linalg. Among these are routines for dealing with matrices and their inverses. The determinant of a matrix can be found:

```
>>> a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)
>>> a
array([[ 4., 2., 0.],
 [ 9., 3., 7.],
 [ 1., 2., 1.]])
>>> np.linalg.det(a)
-53.999999999999993
```

The inverse of a matrix can be found:

```
>>> b = np.linalg.inv(a)
>>> b
array([[ 0.14814815, 0.07407407, -0.25925926],
 [ 0.2037037 , -0.14814815, 0.51851852],
 [-0.27777778, 0.11111111, 0.11111111]])
>>> np.dot(a, b)
array([[ 1.00000000e+00, 5.55111512e-17, 2.22044605e-16],
 [ 0.00000000e+00, 1.00000000e+00, 5.55111512e-16],
 [ 1.11022302e-16, 0.00000000e+00, 1.00000000e+00]])
```

There are many other linear algebra operations which are beyond the scope of this class.

9.4.5 Statistics

In addition to the mean, var, and std functions, NumPy supplies several other methods for returning statistical features of arrays. The median can be found:

```
>>> a = np.array([1, 4, 3, 8, 9, 2, 3], float)
>>> np.median(a)
3.0
```

The correlation coefficient for multiple variables observed at multiple instances can be found for arrays of the form [[x1, x2, ...], [y1, y2, ...], [z1, z2, ...], ...] where x, y, z are different observables and the numbers indicate the observation times:

```
>>> a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
>>> c = np.corrcoef(a)
>>> c
array([[ 1. , 0.72870505],
 [ 0.72870505, 1. ]])
```

Here the return array c[i,j] gives the correlation coefficient for the ith and jth observables. Similarly, the covariance for data can be found:

```
>>> np.cov(a)
array([[ 0.91666667, 2.08333333],
 [ 2.08333333, 8.91666667]])
```

9.4.6 Random Numbers

An important part of any simulation is the ability to draw random numbers. For this purpose, we use NumPy's built-in pseudorandom number generator routines in the sub-module random. The numbers are pseudo random in the sense that they are generated deterministically from a seed number, but are distributed in what has statistical similarities to random fashion[1].

The random number seed can be set:

```
np.random.seed(293423)
```

The seed is an integer value. Any program that starts with the same seed will generate exactly the same sequence of random numbers each time it is run. This can be useful for debugging purposes, but one does not need to specify the seed and in fact, when we perform multiple runs of the same simulation to be averaged together, we want each such trial to have a different sequence of random numbers. If this command is not run, NumPy automatically selects a random seed (based on the time) that is different every time a program is run. An array of random numbers in the half-open interval [0.0, 1.0) can be generated:

```
>>> np.random.rand(5)
array([ 0.40783762, 0.7550402 , 0.00919317, 0.01713451, 0.95299583])
```

The rand function can be used to generate two-dimensional random arrays, or the resize function could be employed here:

```
>>> np.random.rand(2,3)
array([[ 0.50431753, 0.48272463, 0.45811345],
 [ 0.18209476, 0.48631022, 0.49590404]])
>>> np.random.rand(6).reshape((2,3))
```

[1]NumPy uses a particular algorithm called the Mersenne Twister to generate pseudorandom numbers

```
array([[ 0.72915152, 0.59423848, 0.25644881],
 [ 0.75965311, 0.52151819, 0.60084796]])
```

To generate a single random number in [0.0, 1.0),

```
>>> np.random.random()
0.70110427435769551
```

To generate random integers in the range [min, max) use randint(min, max):

```
>>> np.random.randint(5, 10)
9
```

In each of these examples, we drew random numbers form a uniform distribution. NumPy also includes generators for many other distributions, some of which may be introduced later.

9.4.7 Note about operations

In many cases the same array operation can be performed using an operator, a method call, or a function call. For example, both of these expressions compute the *memberwise* sum of two arrays:

```
a + b             # Using an operator
numpy.add(a, b)   # Using a function call
```

and both of these expressions compute the dot product of two arrays:

```
a.dot(b)          # Using a method call
numpy.dot(a, b)   # Using a function call
```

Generally we use an operator if it is available, we use a method call only if an operator is unavailable, and we use a function call only if neither an operator nor a method call is available.

Chapter 10

Basic Plotting

10.1 Matplotlib: Pylab

The standard package for curve plotting in Python is Matplotlib. First we exemplify Matplotlib using matplotlib.pylab, which enables a syntax very close to that of Matlab.

As a first example let us plot the curve $y = t^2e^{-t^2}$ for t values between 0 and 3. First we generate equally spaced coordinates for t, say 51 values (50 intervals). Then we compute the corresponding y values at these points, before we call the plot(t,y) command to make the curve plot.

```python
from matplotlib.pylab import *
def f(t):
    return t**2*exp(-t**2)
t = linspace(0, 3, 51)
y = zeros(len(t))
for i in arange(len(t)):
    y[i] = f(t[i])
# or simply y = f(t), which yields faster and shorter code
plot(t, y)
show()
```

The from "**matplotlib.pylab import** *" command performs two commands in the background: "**from numpy import** *" and import of all Matplotlib commands. We have already mentioned that this is bad practice. However, doing so creates a namespace that resembles Matlab-style syntax.

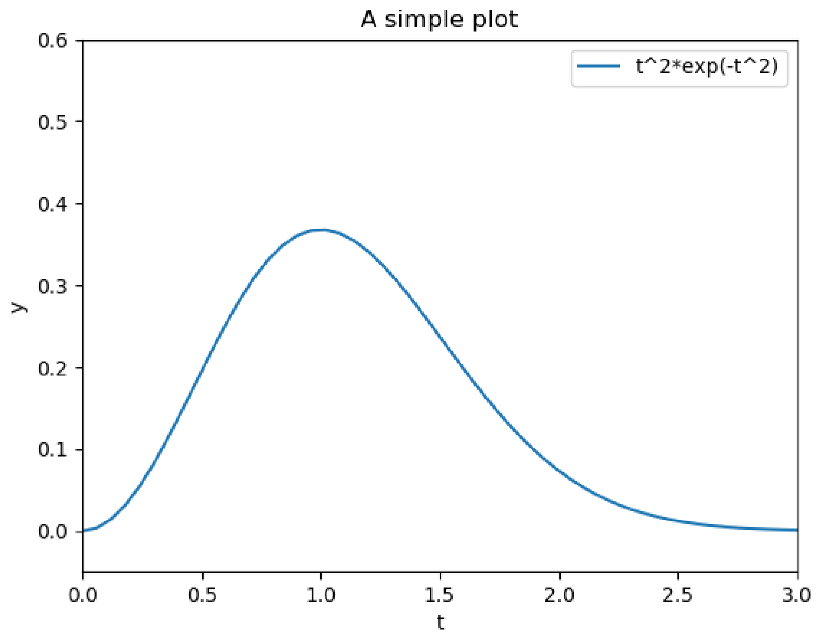
One can generate an image plot in PNG or other image formats. The savefig function saves the plot to files in various image formats:

```python
savefig('exp_plot.png') # produce PNG
```

The x and y axes in curve plots should have labels, here t and y, respectively. Also, the curve should be identified with a label, or legend as it is often called. A title above the plot is also common. In addition, we may want to control the extent of the axes (although most plotting programs will automatically adjust the axes to the range of the data). All such things are easily added after the plot command:

```
xlabel('t')
ylabel('y')
legend(['t^2*exp(-t^2)'])
axis([0, 3, -0.05, 0.6]) # [tmin, tmax, ymin, ymax]
title('A simple plot')
show()
```

The generated image file is shown in figure 10.1.

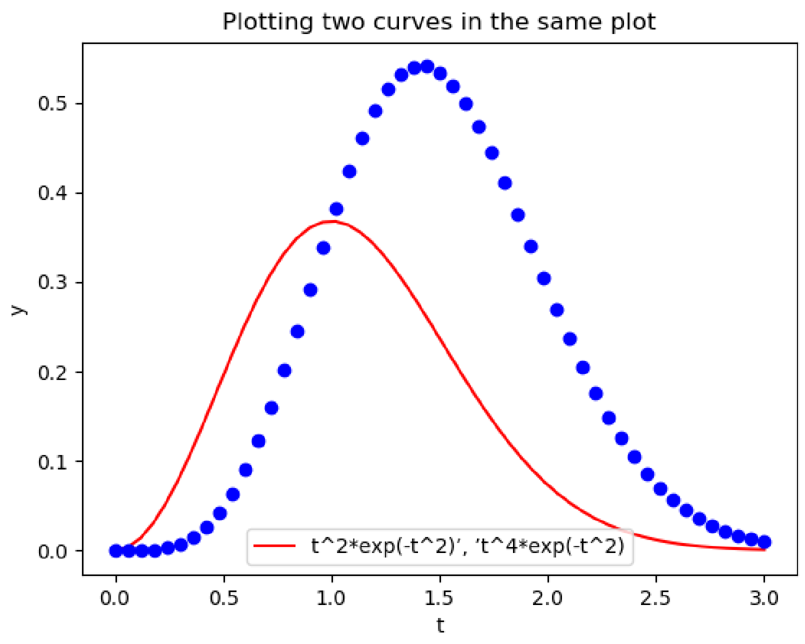


A common plotting task is to compare two or more curves, which requires multiple curves to be drawn in the same plot. Suppose we want to plot the two functions $f_1(t) = t^2e^{-t^2}$ and$f_2(t) = t^4e^{-t^2}$. We can then just issue two plot commands, one for each function:

```
def f1(t):
    return t**2*exp(-t**2)
def f2(t):
    return t**2*f1(t)
t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plot(t, y1, 'r-')
plot(t, y2, 'bo')
xlabel('t')
ylabel('y')
legend(['t^2*exp(-t^2)?, ?t^4*exp(-t^2)'])
title('Plotting two curves in the same plot')
show()
```

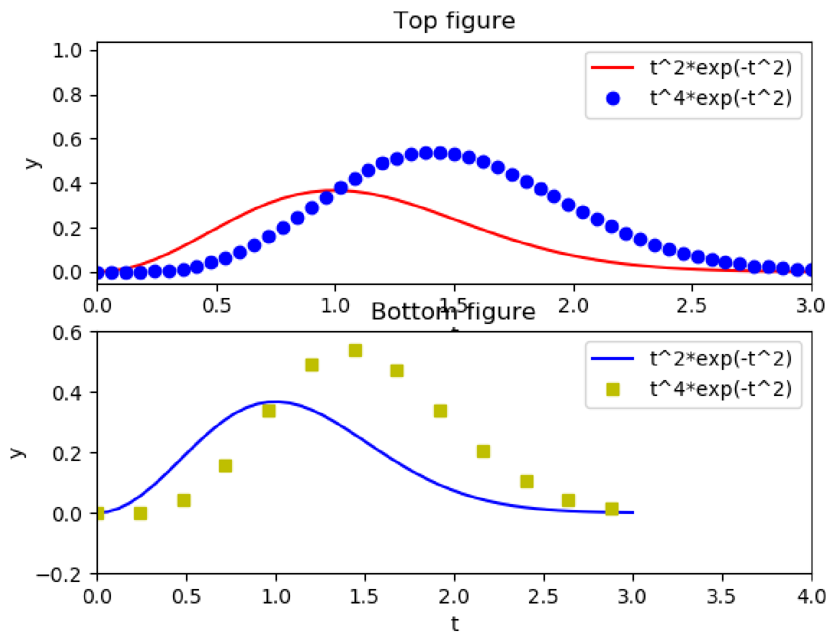
In these plot commands, we have also specified the line type: r- means red (r) line (-), while bo means a blue (b) circle (o) at each data point.The legends for each curve is specified in a list where the sequence of strings correspond to the sequence of plot commands. The generated image file is shown in figure 10.1. We may also put plots together in a figure with r rows and c columns of



plots. The subplot(r,c,a) does this, where a is a row-wise counter for the individual plots.

```
subplot(2, 1, 1)
plot(t, y1, 'r-', t, y2, 'bo')
xlabel('t')
ylabel('y')
axis([t[0], t[-1], min(y2)-0.05, max(y2)+0.5])
legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
title('Top figure')

subplot(2, 1, 2)
t3 = t[::4]
y3 = f2(t3)
plot(t, y1, 'b-', t3, y3, 'ys')
xlabel('t')
ylabel('y')
axis([0, 4, -0.2, 0.6])
legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
title('Bottom figure')
show()
```



10.2 Matplotlib: Pyplot

In this class we do not promote the matplotlib.pylab interface described in the previous section. However, you may use it for simple plots and quick prototyping. Instead, we recommend using the matplotlib.pyplot module. The standard practice is to prefix Numerical Python and Matplotlib functionality by short forms of their package names:

```python
import numpy as np
import matplotlib.pyplot as plt
```

The commands in **matplotlib.pyplot** are similar to those in **matplotlib.pylab**. Most can typically be obtained by prefixing the pylab commands with plt:

```python
plt.xlabel('t')
plt.ylabel('y')
plt.legend(['t^2*exp(-t^2)'])
plt.axis([0, 3, -0.05, 0.6]) # [tmin, tmax, ymin, ymax]
plt.title('A simple plot')
plt.show()
```

Once you have created a basic plot, there are numerous possibilities for fine-tuning the figure, i.e., adjusting tick marks on the axis, inserting text, etc. The Matplotlib website[1] is full of instructive examples on what you can do with this excellent package[2].

[1]https://matplotlib.org/

[2]There are several other alternatives for specialized plotting. However, we will ony use Matplotlib in this class.

Appendix A

Using the Command Line

A.1 Using the Command Line

In CDS 230, you will mainly run Python programs two ways: using the Spyder IDE or using the command line. This is a small guide in using the command line.

The command-line shell, sometimes called the command prompt or the terminal, is a tool that lets you control your computer using only textual commands. It offers a lot of power and simplicity (simplicity is different from ease of use).

Just like with a graphical file browser such as the Finder or Windows Explorer, there is a ”current directory” that you are currently working in. (”Directory” and ”folder” are synonyms.) You can issue commands that operate in that directory, or you can change the current directory.

This guide presents an example transcript of using the shell for Unix (Mac/Linux) and Windows machines. The transcript assumes that the student has already installed the Anaconda Python Distribution, and has created the CDS-230 directory structure as described in the lecture. When you run similar commands, there may be slight differences from the example transcript, such as the number, names, and times of files.

See the section that is relevant to you:

A.1.1 Mac/Linux

Here are most of the commands you will need to use:

- *pwd* - print the absolute pathname of your current working directory
- *cd directory* - change your working directory to the given directory
- *cd ..* - change your working directory to the parent of the current working directory
- *ls* - list the contents of the current directory (”ls” is short for ”list”)
- *mkdir cds-230* - *mkdir* creates a directory named cds-230
- python - run the Python interpreter

- python *program.py* - run the Python program that is stored in the *program.py* file You can open a command-line shell by running the *terminal* program.

In the example below, $ is the prompt at which the user types commands. What follows the $ prompt was printed by the command-line shell.

```
$ pwd
/home/me
$ ls
Desktop    Downloads  Music  Pictures     Public    Templates   Videos
Documents  Dropbox    Old    Programming  Software  Ubuntu One  VirtualBox VMs
$ cd Desktop
$ pwd
/home/me/Desktop
$ ls
cds-230
$ cd cds-230
$ pwd
/home/me/Desktop/cds-230
$ ls
data scripts fall-2019
$ cd scripts
$ pwd
/home/me/Desktop/cds-230/scripts
$ ls
helloworld.py
$ python helloworld.py
Hello world!
```

A.1.2 Windows

Here are most of the commands you will need to use:

- *echo %cd%* - print the absolute pathname of your current working directory
- *cd directory* - change your working directory to the given directory
- *cd ..* - change your working directory to the parent of the current working directory
- *dir* - list the contents of the current directory (”ls” is short for ”list”)
- *mkdir cds-230* - *mkdir* creates a directory named cds-230
- python - run the Python interpreter
- python *program.py* - run the Python program that is stored in the *program.py* file You can open a command-line shell by running the *terminal* program.

You can open a command-line shell by running the cmd program. You should have a Command Prompt shortcut located in the Start Menu, in the Accessories submenu of All Programs, or on the Apps screen for Windows 8. About.com has more detailed instructions about starting the command prompt.

In the example below, `C:\Users\Me>` is the prompt at which the user types commands. What follows the prompt was printed by the command-line shell.

```
C:\Users\Me>echo \%cd\%
C:\Users\Me

C:\Users\Me>dir
 Directory of C:\Users\Me

06/02/2012  08:11 PM    <DIR>          .
06/02/2012  08:11 PM    <DIR>          ..
07/18/2012  05:03 PM    <DIR>          Contacts
01/10/2013  07:24 PM    <DIR>          Desktop
07/18/2012  05:03 PM    <DIR>          Documents
01/09/2013  09:59 PM    <DIR>          Downloads
07/18/2012  05:03 PM    <DIR>          Favorites
07/18/2012  05:03 PM    <DIR>          Links
07/18/2012  05:03 PM    <DIR>          Music
11/28/2012  09:19 PM    <DIR>          Pictures
11/29/2012  01:42 AM    <DIR>          Saved Games
07/18/2012  05:03 PM    <DIR>          Searches
11/27/2012  09:06 PM    <DIR>          Videos

C:\Users\Me>cd Desktop

C:\Users\Me\Desktop>mkdir cds-230
C:\Users\Me\Desktop>dir
 Directory of C:\Users\Me\Desktop

01/10/2013  07:25 PM    <DIR>          .
01/10/2013  07:25 PM    <DIR>          ..
01/10/2013  07:25 PM    <DIR>          cds-230

C:\Users\Me\Desktop>cd cds-230

C:\Users\Me\Desktop>mkdir scripts
C:\Users\Me\Desktop>mkdir data
C:\Users\Me\Desktop\cds-230>dir
 Directory of C:\Users\Me\Desktop\cds-230

01/10/2013  07:25 PM    <DIR>          .
01/10/2013  07:25 PM    <DIR>          ..
01/10/2013  07:25 PM    <DIR>          data
```

```
01/10/2013  07:24 PM    <DIR>          scripts

C:\Users\Me\Desktop\cds-230>cd scripts

C:\Users\Me\Desktop\cds-230\homework2>dir
 Directory of C:\Users\Me\Desktop\cds-230\scripts

01/10/2013  07:24 PM    <DIR>          .
01/10/2013  07:24 PM    <DIR>          ..
01/09/2013  09:26 PM            13   helloworld.py

C:\Users\Me\Desktop\cds-230\scripts>python helloworld.py
Hello world!
```

Appendix B

Computational Problem Solving

Computational problem solving does not simply involve the act of computer programming. It is a process, with programming being only one of the steps. Before a program is written, a design for the program must be developed. And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested. These steps are outlined below.

```
ANALYSIS
                   Clearly understand the problem
                   Know what constitutes a solution
```

```
DESIGN
                   Determine what type of data is needed
                   Determine how the data is to be structured
                   Find another design appropriate algorithm
```

```
IMPLEMENTATION
                   Represent data within programming language
                   Implement algorithms in programming language
```

```
TESTING
                   Test the program on a selected set of problem instances
                   Correct and understand the causes of any errors found
```

1. ANALYSIS

   (a) Understanding the problem. Once a problem is clearly understood, the fundamental computational issues for solving it can be determined.

   (b) Knowing what constitutes a solution. For some problems, there is only one solution. For others, there may be a number (or infinite number) of solutions. Thus, a program may be stated as finding

- A solution
- An approximate solution
- A best solution
- All solutions

2. DESIGN

   (a) Describing the data needed. This, of course, depends on the problem at hand. We can use a list, a table, a matrix, etc.

   (b) Describing the Needed Algorithms. For some problems, there is only one solution. When solving a computational problem, either suitable existing algorithms may be found or new algorithms must be developed. Algorithms that work well in general but are not guaranteed to give the correct result for each specific problem are called *heuristic algorithms.*

3. IMPLEMENTATION Design decisions provide general details of the data representation and the algorithmic approaches for solving a problem. The details, however, do not specify which programming language to use, or how to implement the program. That is a decision for the implementation phase. Since we are programming in Python, the implementation needs to be expressed in a syntactically correct and appropriate way, using the instructions and features available in Python.

4. TESTING Software testing is a crucial part of software development. Testing is done incrementally as a program is being developed, when the program is complete, and when the program needs to be updated.

Appendix C

References

Tutorials

Tutorials for beginners:
`https://www.w3schools.com/PYTHON/python_lists.asp`
`https://www.tutorialspoint.com/python/`

A Python tutorial from the official Python website:
`https://docs.python.org/3/tutorial//`

For exact syntax and semantics of the Python language:
`http://docs.python.org/3/`

Reference manual of the standard library:
`http://devdocs.io/python`

Online Tools

The Python Online Tutor allows you to visualize execution of Python code.
`http://people.csail.mit.edu/pgbovine/python/tutor.html`

Online Python interpreter: Just in case Jupyter Notebook is not enough.
`https://www.onlinegdb.com/online_python_interpreter`

Modeling and Simulation

This book has material similar in spirit to CDS230 but slightly different approach:
`http://greenteapress.com/wp/modsimpy`