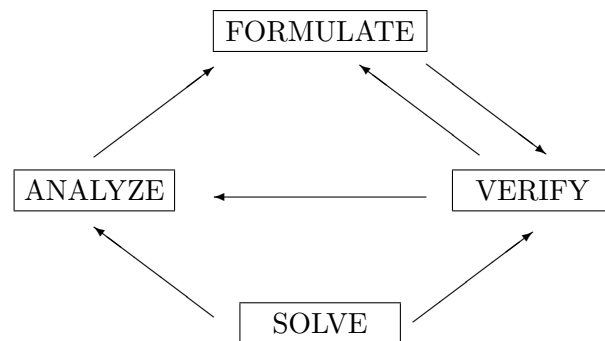


George Mason University

CDS 230

Fall 2019

Carlos Cruz



Modeling and Simulation I

Course Notes

# Contents

<b>1</b>	<b>Setup</b>	<b>5</b>
1.1	Your first program . . . . .	5
1.1.1	What is a Python program? . . . . .	5
1.1.2	Executing a Python Program . . . . .	6
1.1.3	Python interpreter vs. Python program . . . . .	6
1.1.4	Errors . . . . .	6
1.1.5	References . . . . .	7
1.1.6	Programming Style . . . . .	7
<b>2</b>	<b>Variables and Data Types</b>	<b>8</b>
2.1	Fundamentals . . . . .	8
2.1.1	Operators . . . . .	8
2.1.2	Summary of Python Arithmetic Operators . . . . .	8
2.1.3	Python Expressions . . . . .	8
2.2	Variables . . . . .	10
2.2.1	Assignment statements . . . . .	10
2.3	Built-in Data Types . . . . .	11
2.3.1	Fundamental Types . . . . .	11
2.4	Formatting Text and Numbers . . . . .	12
2.4.1	Number Formatting . . . . .	13
2.4.2	string.format() basics . . . . .	13
2.5	Examples . . . . .	14
<b>3</b>	<b>Control Flow</b>	<b>19</b>
3.1	Boolean Expressions (Conditions) . . . . .	19
3.1.1	Conditional Operators . . . . .	19
3.1.2	Logical Operators . . . . .	20
3.2	Conditionals . . . . .	21
3.2.1	if Statements . . . . .	21
3.3	Loops . . . . .	24

3.3.1	Indefinite Iteration: while Loops . . . . .	24
3.3.2	for Loops . . . . .	26
<b>4</b>	<b>Lists and Tuples</b>	<b>28</b>
4.1	Sequence Types . . . . .	28
4.1.1	Lists . . . . .	28
4.1.2	Iterating Over List Elements vs. List Index Values . . . . .	31
4.1.3	List Comprehensions . . . . .	31
4.1.4	Traversing Multiple Lists Simultaneously . . . . .	32
4.1.5	Tuples . . . . .	32
4.1.6	Nested Data Structures . . . . .	33
4.1.7	Copying Lists . . . . .	33
<b>5</b>	<b>Dictionaries and Sets</b>	<b>35</b>
5.1	Sets . . . . .	35
5.1.1	Properties . . . . .	35
5.1.2	Initialization . . . . .	35
5.1.3	Set operations . . . . .	36
5.1.4	Modifying a Set . . . . .	36
5.2	Dictionaries . . . . .	37
5.2.1	Properties . . . . .	37
5.2.2	Initialization . . . . .	38
5.2.3	Dictionary Operations . . . . .	38
<b>6</b>	<b>Functions</b>	<b>41</b>
6.1	Functions . . . . .	41
6.1.1	Defining Functions . . . . .	42
6.1.2	Calling Functions . . . . .	43
6.1.3	Parameter Passing . . . . .	44
6.1.4	Variable Scope . . . . .	46
6.1.5	Functions as Arguments to Functions . . . . .	47
6.1.6	The Main Program . . . . .	48
6.1.7	Lambda Functions . . . . .	49
6.2	A Bioinformatics Example . . . . .	49
6.2.1	Counting Letters in DNA Strings . . . . .	49
<b>A</b>	<b>Using the Command Line</b>	<b>56</b>
A.1	Using the Command Line . . . . .	56
A.1.1	Mac/Linux . . . . .	56
A.1.2	Windows . . . . .	57

<b>B Computational Problem Solving</b>	<b>60</b>
<b>C References</b>	<b>62</b>

# 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<sup>1</sup>. Then the Python **interpreter** directs your computer to follow the instructions expressed in that language. Note that the **interpreter** is a loop<sup>2</sup> 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.

---

<sup>1</sup>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.

<sup>2</sup>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 you 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

<code>+</code> Addition	Adds values on either side of the operator.
<code>-</code> Subtraction	Subtracts right hand operand from left hand operand.
<code>*</code> Multiplication	Multiplies values on either side of the operator
<code>/</code> Division	Divides left hand operand by right hand operand
<code>//</code> Floor Division	Returns integer part of the quotient
<code>%</code> modulo	Divides left hand operand by right hand operand and returns remainder
<code>**</code> 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**:

```
1 2
2 1 + 2
3 2 **12
4 1/-12
5 (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**<sup>1</sup>. So, the expression  $(72 - 32)/9 * 5$  is evaluated as follows:

```
1 (72 - 32)/9*5
2 (40)/9*5
3 40/9*5
4 4.44*5
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**<sup>2</sup>.

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!)

```
1 5/2
2 5/2.0
3 5.0/2
4 7*(1/2)
5 7*(1/2.0)
6 5**2
7 5.0**2
8 5**2.0
9 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!

---

<sup>1</sup>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.

<sup>2</sup>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 discussed how one can assign values to a variable. Let's look at that in more detail. Consider the following series of statements<sup>3</sup>:

```
1 In [1]: x = 2
2 In [2]: print(id(x), x)
3 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?

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

After two consecutive assignments the *ids* 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.

---

<sup>3</sup>*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 `+`.

```
1 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:

```
1 x = 1
2 y = 2
3 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:

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

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

```
1 x = 1.5
2 print(x, type(x))
3 (1.5, <type 'float'>)
4 x = int(x)
5 print(x, type(x))
6 (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:

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

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

```
1 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<sup>4</sup> 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:

```
1 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.

```
1 s1 = "Python is {}".format("a very popular language")
2 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.

---

<sup>4</sup>There are many more ways. These are the ones we'll use in this class. For more information see the Python documentation.

```

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

```

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

```

1 t = 0.6
2 y = 1.23456
3 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:**

```

1 # Convert from Fahrenheit degrees to Celsius degrees
2 F = 70
3 C = (5.0/9)*(F - 32)
4 print(C)
5 Out[]: 21.111111111111111

```

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

**Solution:**

```

1 from math import sin, cos, pi
2 x = pi/4
3 one = sin(x)**2 + cos(x)**2
4 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?

```
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
6 print(x1, x2)
```

Upon running the program we will get the following output:

```
1 1 a = 2; b = 1; c = 2
2 2 from math import sqrt
3 > 3 q = sqrt(b*b - 4*a*c)
4 4 x1 = (-b + q)/2*a
5 5 x2 = (-b - q)/2*a
6 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 trajectory of a ball thrown at an angle  $\theta$  with the horizontal ball will follow a *trajectory*  $y = f(x)$  through the air, where

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

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  $\theta$  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/s<sup>2</sup>;  $x, y$ , and  $y_0$  are measured in meters; and  $\theta$  in degrees. The program has naturally four parts: initialization of input data, import of functions and  $\pi$  from math, conversion of  $v_0$  and  $\theta$  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:

```
1 g = 9.81      # m/s**2
2 v0 = 15       # km/h
3 theta = 60    # degrees
4 x = 0.5       # m
5 y0 = 1        # m
6
```

```

7 print("""\
8 v0      = {:.1f} km/h
9 theta   = {:d} degrees
10 y0      = {:.1f} m
11 x       = {:.1f} m\
12 """).format(v0, theta, y0, x)
13 )
14
15 from math import pi, tan, cos
16 # Convert v0 to m/s and theta to radians
17 v0 = v0/3.6
18 theta = theta*pi/180
19
20 y = x*tan(theta) - 1/(2*v0**2)*g*x**2/((cos(theta))**2) + y0
21
22 print('y      = {:.1f} m' .format(y))
23 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:

```
1 year=1992
2 month=5
3 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,

```
1 numsecs_day = (hours per day) * (mins per hour) * (secs per minute)
2 numsecs_year = (days per year) * numsecs_day
3 avg_numsecs_year = (4 * numsecs_year) + numsecs_day // 4
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.

```
1 numsecs_1900_to_dob = (year_birth - 1900) * avg_numsecs_year +
2     (month_birth - 1) * avg_numsecs_month +
3     (day_birth * numsecs_day)
4 numsecs_1900_to_today = (current_year - 1900) * avg_numsecs_year +
5     (current_month - 1) * avg_numsecs_month +
6     (current_day * numsecs_day)
7 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`.

```
1 import datetime
2
3 # Inputs
4 month_birth = int(input('Enter month born (1-12): '))
5 day_birth = int(input('Enter day born (1-31): '))
6 year_birth = int(input('Enter year born (4 digit): '))
7
8 # Get current time
9 current_month = datetime.date.today().month
```

```

10 current_day = datetime.date.today().day
11 current_year = datetime.date.today().year
12
13 # test output:
14 print("Input is {} {} {}:".format(month_birth, day_birth, year_birth
    ))
15 print("Current date is {} {} {}:".format(current_month, current_day,
    current_year))
16
17 # Main algorithm
18 numsecs_day = 24*60*60
19 numsecs_year = 365*numsecs_day
20
21 avg_numsecs_year = (4 * numsecs_year) + numsecs_day) // 4
22 avg_numsecs_month = avgnumsecs_year // 12
23
24 numsecs_1900_to_dob = (year_birth - 1900) * avg_numsecs_year + \
25     (month_birth - 1) * avg_numsecs_month + \
26     (day_birth * numsecs_day)
27 numsecs_1900_to_today = (current_year - 1900) * avg_numsecs_year + \
28     (current_month - 1) * avg_numsecs_month + \
29     (current_day * numsecs_day)
30 age_in_secs = numsecs_1900_to_today - numsecs_1900_to_dob
31 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:

```

1 print( True and True )      # prints True
2 print( True and False )    # prints False
3 print( 3 < 4 and 10 < 12 )  # prints True
4 print( 3 < 4 or 12 < 10 )   # prints True
5 print( 4 < 3 or 12 < 10 )   # prints False
6 print( (4 < 3 and 12 < 10) or 7 == 7 ) # prints True
7 print(10 < 0 and not 10 > 2) # prints False
8 '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:

```

1 z = 1 > 2
2 print(z)
3 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:

```

1 from math import pi, sin
2
3 print( pi )                # prints 3.14159265359
4 print( sin(pi) )           # prints 1.22464679915e-16
5 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  $\pi$ . 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:

```
1 bool("hello")
2 bool(1 and 1)
3 bool(0 and "test")
4 bool(False or 1)
5 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

```
1 if condition:                                # HEADER
2     Python code that runs iff condition is True    # CLAUSE
3     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**<sup>1</sup> must same the same indentation<sup>2</sup>. Example:

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

### Binary Selection:

```
1 if condition:
2     Python code that runs iff condition is True
3 else:
4     Python code that runs iff condition is False
5     ...again, indentation is important
```

This control structure divides the flow in two depending on whether the control condition is True or False. Example<sup>3</sup>:

```
1 n = int(input('Enter a number: '))
2 if n % 2 == 0:
3     print ("Number is even")
4 else:
5     print ("Number is odd")
6 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<sup>4</sup>:

```
1 if condition1:
2     Python code that runs iff condition1 is True
3 elif condition2:
4     Python code that runs iff condition2 is True
5 elif condition3:
6     Python code that runs iff condition3 is True
7 else:
8     Python code that runs iff conditions 1-3 are False
```

---

<sup>1</sup>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.

<sup>2</sup>In Python, 4 spaces is the standard

<sup>3</sup>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**

<sup>4</sup>Note the else clause is optional.

Example:

```
1 n = int(input('Enter a number: '))
2 if n < 0:
3     print ('n is negative')
4 elif n > 0:
5     print ('n is positive')
6 else:
7     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:**

```
1 if condition1:
2     Python code that runs iff condition1 is True
3 else:
4     Python code that runs iff condition1 is False
5     if condition2:
6         Python code that runs iff condition2 is True
7     else:
8         Python code that runs iff condition2 is False
```

Example:

```
1 n = int(input('Enter a number: '))
2 if n > 0:
3     print ('n is positive')
4     if n % 2 == 0:
5         print ('...and also even')
6     else:
7         print ('...and also odd')
8 else:
9     if n == 0:
10        print ('n is zero')
11    else:
12        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<sup>5</sup> 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:

```
1 while condition:      # HEADER
2     # 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.

```
1 C = -20      # starting value for C
2 dC = 5       # increment of C in loop
3 while C <= 40:    # loop heading with condition
4     F = (9/5)*C + 32    # Conversion from C to F
5     print(C,F)        # Result
6     C = C + dC        # Increment
7 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:

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

---

<sup>5</sup>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<sup>6</sup>. 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:

```
1 C += dC # equivalent to C = C + dC
2 C -= dC # equivalent to C = C - dC
3 C *= dC # equivalent to C = C*dC
4 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.

```
1 while condition:
2     # some code
3     if condition:
4         break # breaks out of loop
```

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

```
1 x = 1
2 while True:
3     if x % 11 == 0 and x % 12 == 0:
4         break
5     x = x + 1
6 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.

```
1 while condition:
2     # some code
3     if condition:
4         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

---

<sup>6</sup>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

```

1 x = 1
2 while x <=100:
3     x += 1
4     if x % 3 ==0 or x % 5 == 0:
5         continue
6         #no more code is executed, we go to the next number
7     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:

```

1 for <var> in <iterable>:
2     <statement(s)>

```

Here <iterable> is a collection of objects<sup>7</sup>. 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

```

1 x = range(5)
2 print(x)
3 range(0, 5)
4 print(type(x))
5 <class 'range'>
6 # Then one can loop:
7 for i in x:
8     print(x)
9 0

```

---

<sup>7</sup>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!

```
10 1
11 2
12 3
13 4
14 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<sup>1</sup>, 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:

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

Note `mixed_list` contains mixed types, including other lists and can contain much more<sup>2</sup>. 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.

---

<sup>1</sup>As a general rule, data structures are objects that contain a possibly large number of other objects.

<sup>2</sup>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  $0 \dots n - 1$ , where  $n$  is the number of elements in the list.

```
1 my_list = [2, 3, 5]
2 my_list[0] # first element in my_list
3 2
4 my_list[1]
5 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**<sup>3</sup>.

```
1 my_list.append(7) # adds 7 to end of my_list
2 my_list.insert(1,0) # inserts element 0 in position 1
3 print(my_list)
4 [2, 0, 3, 5, 7]
5 my_list.pop() # removes last element in list
6 print(my_list)
7 [2, 0, 3, 5]
8 del my_list[1] # deletes second element
9 print(my_list)
10 [2, 3, 5]

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

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

---

<sup>3</sup>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.

```

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

```

You can specify a stride - either positive or negative:

```

1 primes[0:6:2] # Here, 2 is the stride prints
2 [2, 5, 11]
3 primes[6:0:-2]
4 [17, 11, 5]

```

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

```

1 primes[::-1]
2 [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:

```

1 stuff = ['this', 'that', 'these', 'those']
2 'this' in stuff
3 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:

```

1 stuff = ['this', 'that', 'these', 'those']
2 'python' not in stuff
3 True
4 'python' in stuff
5 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 **\***.

```

1 stuff = ['this', 'that', 'these', 'those']
2 stuff + ['them']
3 ['this', 'that', 'these', 'those', 'them']
4 stuff*2
5 ['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.

```

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

```

```

4 max(primes) # 31
5 len(primes) # 11
6 len(stuff) # 4
7 min(stuff) # 'that'
8 max(stuff) # 'those'
9 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.

```

1 # Loop over list elements
2 primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
3 for n in primes:
4     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.

```

1 nums = [10, 20, 30, 40, 50, 60]
2 # Iterate over elems of list
3 for k in nums:
4     sum = sum + k
5 # Loop over index values
6 for k in range(len(nums)):
7     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.

```

1 nums = [10, 20, 30, 40, 50, 60]
2 k = 0
3 wanted = 40
4 found = False
5 while k < len(nums) and not found:
6     if nums[k] == wanted:
7         found = True
8     else:
9         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

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

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

```
1 my_nums = [i*0.5 for i in range(10)]
2
3 Cdegrees = [-5 + i*0.5 for i in range(10)]
4 Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
5
6 S = [x**2 for x in range(10)]
7 M = [x for x in S if x % 2 == 0]
8
9
10 In []: my_nums
11 Out[]: [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
12
13 In []: Cdegrees
14 Out[]: [-5.0, -4.5, -4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -0.5]
15
16 In []: Fdegrees
17 Out[]: [23.0, 23.9, 24.8, 25.7, 26.6, 27.5, 28.4, 29.3, 30.2, 31.1]
18
19 In []: S
20 Out[]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
21
22 In []: M
23 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

```
1 for list1, list2, list3... in zip(list1, list2, list3, ...):
2     # 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:

```
1 empty_tuple = ()
2 # or
3 empty_list = tuple() # list() constructor
4 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:

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

```
1 list1 = [10, 20, 30, 40]
2 list2 = list1
3 list1[0] = 5
4 print(list1)
5 [5, 20, 30, 40] # change made in list1
6 print(list2)
7 [5, 20, 30, 40] # change in list1 causes a change in list2
```

Knowing that variables *list1* and *list2* 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,

```
1 list1 = [10, 20, 30, 40]
2 list2 = list(list1)
3 list1[0] = 5
4 print(list1)
5 [5, 20, 30, 40] # change made in list1
6 print(list2)
7 [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.

```
1 # importing "copy" for copy operations
2 import copy
3 list1 = [1, 2, [3,5], 4]
4 # using copy to shallow copy
5 shallow_copy = copy.copy(list1)
6 # using deepcopy to deep copy
7 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:

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

Unfortunately you cannot use `{}` to express empty set.

Construct from a list (or a tuple or a string):

```
1 empty = set() # constructor
2 odd = set([1,3,5])
3 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: <code>in</code>	<code>4 in primes</code> $\Rightarrow$ False
union $\cup$	Python: <code> </code>	<code>odd   primes</code> $\Rightarrow$ {1,2,3,5}
intersection $\cap$	Python: <code>&amp;</code>	<code>odd &amp; primes</code> $\Rightarrow$ {3,5}
difference $-$	Python: <code>-</code>	<code>odd - primes</code> $\Rightarrow$ {1}
symmetric difference	Python: <code>^</code>	<code>odd ^ primes</code> $\Rightarrow$ {1,2}

do iteration over stes

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

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

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

### 5.1.4 Modifying a Set

Add element to a set

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

Remove element from a set

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

Remove an arbitrary element from a set

```
1 primes.pop()
```

Try the following:

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

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

```

1 # Using lists and loops:
2 list1 = [1,2,3,4,5,6,7,8]
3 list2 = [2,4,6,8]
4 out1 = []
5 for i in list2:
6     if i in list1:
7         out1.append(i)
8
9 # Using list comprehensions would reduce the last 4 lines into 1:
10 out1 = [i for i in list2 if i in list1]

```

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

```

1 list1 = [1,2,3,4,5,6,7,8]
2 list2 = [2,4,6,8]
3 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 ^ 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<sup>1</sup>.

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

---

<sup>1</sup>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.

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

Examples of `dict()` syntax:

```
1 # Temperatures:  
2 daily_temps = {'sun': 68.8,  
3               'mon': 70.2,  
4               'tue': 67.2,  
5               'wed': 71.8,  
6               'thur': 73.2,  
7               'fri': 75.6,  
8               'sat': 74.0}  
9 # Population of cities (in millions)  
10 population = {"Chicago":2.7, "New York":8.17, "Rome":2.87,  
11              "Paris":2.24, "London":8.78}  
12 # HTTP response status codes  
13 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<sup>2</sup>. Let's see how we can use `daily_temps`:

```
1 if daily_temps['sun'] > daily_temps['sat']:  
2     print('Sunday was the warmer weekend day')  
3 else:  
4     if daily_temps['sun'] < daily_temps['sat']:  
5         print('Saturday was the warmer weekend day')  
6     else:  
7         print('Saturday and Sunday were equally warm')
```

---

<sup>2</sup>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.

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

### Accessing dictionary elements:

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

### Dictionary functions:

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

### Iterating through a dictionary:

```
1 atomic_number = {"H":1, "Fe":26, "Au":79}
2
3 # Print out all the keys:
4 for element_name in atomic_number.keys():
5     print(element_name)
6 # Print out all the values:
7 for element_number in atomic_number.values():
8     print(element_number)
9 # Print out the keys and the values
10 for (element_name, element_number) in atomic_number.items():
11     print("name:", element_name, "number:", element_number)
```

### Modifying a dictionary:

```
1 us_wars1 = {
2     "Revolutionary" : [1775, 1783],
```

```

3     "Mexican" : [1846, 1848],
4     "Civil" : [1861, 1865] }
5 us_wars1["WWI"] = [1917, 1918] # add mapping
6 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}

```

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

```

### Try these on your own:

```

1 squares = { 1:1, 2:4, 3:9, 4:16 }
2 squares[3] + squares[3]
3 squares[3 + 3]
4 squares[2] + squares[2]
5 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<sup>1</sup>. 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<sup>2</sup> and we have met several so far.

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

---

<sup>1</sup>Also known as the Don't Repeat Yourself (DRY) principle).

<sup>2</sup>Built-in functions are always available and do not need to be imported. Other functions come from modules, like **math** or **random**.

```

6 int(input("enter integer: "))
7 range(1,5)
8
9 # functions from modules
10 import math
11 math.sin(math.pi)
12 import random
13 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:

```

1 def a_good_name(arg1, arg2, arg3...): # HEADER
2     # Python statements
3     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<sup>3</sup>. 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,  $f(2) = 2x = 4$ :

```

1 def f(x): # x is a parameter
2     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.

---

<sup>3</sup>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.

```
1 def print_hello():
2     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:

```
1 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:

```
1 def square(x):    # x is a parameter
2     return x * x
3 sq = square(2)    # 2 is an argument, function returns sq=4
4
5 def my_sum(x, y):
6     return x + y
7 sum = my_sum(2,3) # returns sum=5
8
9 def quadratic_roots(a, b, c):
10    # some code to solve quadratic formula
11    return x1, x2
12 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:

```
1 def print_hello():
2     print("Hello, world")
3
4 # calling print_hello()
5 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.

```
1 def line(x0, y0, x1, y1):
2     """
3     Compute the coefficients a and b in the mathematical
4     expression for a straight line  $y = a*x + b$  that goes
5     through two points (x0, y0) and (x1, y1).
6     x0, y0: a point on the line (floats).
7     x1, y1: another point on the line (floats).
8     return: coefficients a, b (floats) for the line (y=a*x+b). """
9     a = (y1 - y0)/float(x1 - x0)
10    b = y0 - a*x0
11    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.,

```
1 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):

```
1 def ordered(n1, n2): # n1, n2 are parameters
2     return n1 < n2 # returns either True or False
3
4 num1 = int(input('Enter num1: '))
5 num2 = int(input('Enter num2: '))
6
7 if ordered(num1, num2): # num1, num2 are arguments
8     print('First number is smaller')
9 else:
10    if ordered(num2, num1):
11        print('First number is larger')
12    else:
13        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.

```
1 def mortgage_rate(amount, rate, term):
2     # some calculations
3     return some_value
4 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:

```
1 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:

```
1 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:

```
1 def mortgage_rate(amount, rate, term=20):
2     # some calculations
3     return some_value
4 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:

```
1 def addup(first, last, incr=-1):
2     if first > last:
3         sum = -1
4     else:
5         sum = 0
6         for i in range(first, last+1, incr):
7             sum = sum + i
8     return sum
9 addup(1, 10)
10 addup(1, 10, 2)
11 addup(first=-1, last = -10)
12 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

```
1 def func1():
2     n = 10
3     print('func1 ',n)
4 def func2():
5     n = 20
6     print('func2 ',n)
7     func1()
8     print('func2 ',n)
9 In [1]: func2()
10 func2  20
11 func1  10
12 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<sup>4</sup>.

### 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:

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

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

```
1 def diff2(f, x, h=1E-6):
2     """
3     approximation of the second-order derivative of a function
4     """
5     r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
6     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

```
1 # This is the function that we want to take the derivative of
2 def g(t):
3     return t**3
4 t = 1.0
5 d2g = diff2(g, t)
6 print("g({})={:.8f}".format(t, d2g))
7 # prints
8 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:

---

<sup>4</sup>For this reason, the use of global variables is generally considered to be bad programming style.

```

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

```

1 from math import exp, sin, pi    # in main
2 def f(x):                        # in main
3     e = exp(-0.1*x)
4     s = sin(6*pi*x)
5     return e*s
6 x= 2                             # in main
7 y= f(x)                          # in main
8 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:

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

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

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

In general,

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

can be written as

```
1 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:

```
1 def count_str0(dna, base):
2     dna = list(dna) # convert string to list of letters
3     i = 0 # counter
4     for c in dna:
5         if c == base:
6             i += 1
7     return i
8 dna = "ATGCGGACCTAT"
9 base = "C"
10 count = count_str0(dna, base)
11 print("{} appears {:d} times in {}".format(base, count, 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.<sup>5</sup> 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):

```
1 def count_str1(dna, base):
2     i = 0 # counter
3     for c in dna:
4         if c == base:
5             i += 1
6     return i
7 dna = "ATGCGGACCTAT"
8 base = "C"
9 count = count_str1(dna, base)
10 print("{} appears {:d} times in {}".format(base, count, dna))
```

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

```
1 def count_str2(dna, base):
2     i = 0 # counter
3     for j in range(len(dna)):
4         if dna[j] == base:
5             i += 1
6     return i
7 dna = "ATGCGGACCTAT"
8 base = "C"
9 count = count_str2(dna, base)
10 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)?

---

<sup>5</sup>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:

```

1 def count_str3(dna, base):
2     found = [] # matches for base in dna: found[i]=True if dna[i]==
3                 base for c in dna:
4     for c in dna:
5         if c == base:
6             found.append(True)
7         else:
8             found.append(False)
9     return sum(found) # note we are using sum here
10 dna = "ATGCGGACCTAT"
11 base = "C"
12 count = count_str3(dna, base)
13 print("{} appears {:d} times in {}".format(base, count, dna))

```

Finally we can use boolean values directly, as follows:

```

1 def count_str4(dna, base):
2     found = []
3     for c in dna:
4         found.append(c == base)
5     return sum(found) # note we are using sum here
6 dna = "ATGCGGACCTAT"
7 base = "C"
8 count = count_str4(dna, base)
9 print("{} appears {:d} times in {}".format(base, count, dna))

```

Finally, let's try Python's library.

```

1 def count_str5(dna, base):
2     return dna.count(base)
3 dna = "ATGCGGACCTAT"
4 base = "C"
5 count = count_str5(dna, base)
6 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:

```
1 N=1000000
2 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:

```
1 import random
2
3 def generate_string(N, alphabet='ACGT'):
4     my_list = []
5     for i in range(N):
6         my_list.append(random.choice(alphabet))
7     return ''.join(my_list) # this returns a string
8 # test
9 print(generate_string(10)) # will be "random"
10 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:

```
1 import time
2 ...
3 t0 = time.clock()
4 # do stuff
5 t1 = time.clock()
6 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:

```
1 import time
2 functions = [count_str0,
3             count_str1,
4             count_str2,
5             count_str3,
6             count_str4,
7             count_str5]
8 timings = [] # timings[i] holds CPU time for functions[i]
9 for function in functions:
10     t0 = time.process_time()
```

```

11     function(dna, 'A')
12     t1 = time.process_time()
13     cpu_time = t1 - t0
14     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:

```

1 for cpu_time, function in zip(timings, functions):
2     print ('{f:<9s}: {cpu:.2f} s'.format(f=function.__name__, cpu=
        cpu_time))
3
4 count_str0: 0.92 s
5 count_str1: 0.85 s
6 count_str2: 1.49 s
7 count_str3: 2.25 s
8 count_str4: 2.13 s
9 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.

# 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.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](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>