George Mason University

CDS 230

Fall 2019

Carlos Cruz

$$\begin{tikzcd}
 & \boxed{\text{FORMULATE}} \arrow[dr, shift left] & \\
\boxed{\text{ANALYZE}} \arrow[ur] & & \boxed{\text{VERIFY}} \arrow[ul, shift left] \arrow[ll] \\
 & \boxed{\text{SOLVE}} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Modeling and Simulation I

Course Notes

Contents

Chapter 1

Setup

1.1 Your first program

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

1.1.1 What is a Python program?

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

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

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

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

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

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

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

1.1.2 Executing a Python Program

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

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

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

```
$ python hello.py
```

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

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

1.1.3 Python interpreter vs. Python program

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

1.1.4 Errors

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

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

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

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

1.1.5 References

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

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

1.1.6 Programming Style

One final item that deserves some elaboration is programming style.

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

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

Chapter 2

Variables and Data Types

2.1 Fundamentals

2.1.1 Operators

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

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

2.1.2 Summary of Python Arithmetic Operators

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

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

2.1.3 Python Expressions

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

```
operands.
```

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

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

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

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

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

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

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

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

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

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

2.2 Variables

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

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

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

2.2.1 Assignment statements

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

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

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

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

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

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

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

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

2.3 Built-in Data Types

2.3.1 Fundamental Types

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

Objects

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

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

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

Integers

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

Floats

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

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

Strings

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

```python
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:

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

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

Booleans

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

isinstance

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

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

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

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

2.4 Formatting Text and Numbers

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

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

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

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

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

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

2.4.1 Number Formatting

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

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

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

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

2.4.2 string.format() basics

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

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

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

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

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

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

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

2.5 Examples

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

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

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

Solution:

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

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

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

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

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

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

Upon running the program we will get the following output:

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

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

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

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

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

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

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

```

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

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

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

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

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

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

The Problem

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

Problem Analysis

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

Program Design

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

Data Description

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

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

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

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

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

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

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

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

```
import datetime

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

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

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

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

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

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

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

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

Appendix A

Using the Command Line

A.1 Using the Command Line

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

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

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

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

See the section that is relevant to you:

A.1.1 Mac/Linux

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

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

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

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

```
$ pwd
/home/me
$ ls
Desktop    Downloads  Music  Pictures     Public    Templates   Videos
Documents  Dropbox    Old    Programming  Software  Ubuntu One  VirtualBox VMs
$ cd Desktop
$ pwd
/home/me/Desktop
$ ls
cds-230
$ cd cds-230
$ pwd
/home/me/Desktop/cds-230
$ ls
data scripts fall-2019
$ cd scripts
$ pwd
/home/me/Desktop/cds-230/scripts
$ ls
helloworld.py
$ python helloworld.py
Hello world!
```

A.1.2 Windows

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

- *echo %cd%* - print the absolute pathname of your current working directory
- *cd directory* - change your working directory to the given directory
- *cd ..* - change your working directory to the parent of the current working directory
- *dir* - list the contents of the current directory (”ls” is short for ”list”)
- *mkdir cds-230* - *mkdir* creates a directory named cds-230
- python - run the Python interpreter
- python *program.py* - run the Python program that is stored in the *program.py* file You can open a command-line shell by running the *terminal* program.

You can open a command-line shell by running the cmd program. You should have a Command Prompt shortcut located in the Start Menu, in the Accessories submenu of All Programs, or on the Apps screen for Windows 8. About.com has more detailed instructions about starting the command prompt.

In the example below, `C:\Users\Me>` is the prompt at which the user types commands. What follows the prompt was printed by the command-line shell.

```
C:\Users\Me>echo \%cd\%
C:\Users\Me

C:\Users\Me>dir
 Directory of C:\Users\Me

06/02/2012  08:11 PM    <DIR>          .
06/02/2012  08:11 PM    <DIR>          ..
07/18/2012  05:03 PM    <DIR>          Contacts
01/10/2013  07:24 PM    <DIR>          Desktop
07/18/2012  05:03 PM    <DIR>          Documents
01/09/2013  09:59 PM    <DIR>          Downloads
07/18/2012  05:03 PM    <DIR>          Favorites
07/18/2012  05:03 PM    <DIR>          Links
07/18/2012  05:03 PM    <DIR>          Music
11/28/2012  09:19 PM    <DIR>          Pictures
11/29/2012  01:42 AM    <DIR>          Saved Games
07/18/2012  05:03 PM    <DIR>          Searches
11/27/2012  09:06 PM    <DIR>          Videos

C:\Users\Me>cd Desktop

C:\Users\Me\Desktop>mkdir cds-230
C:\Users\Me\Desktop>dir
 Directory of C:\Users\Me\Desktop

01/10/2013  07:25 PM    <DIR>          .
01/10/2013  07:25 PM    <DIR>          ..
01/10/2013  07:25 PM    <DIR>          cds-230

C:\Users\Me\Desktop>cd cds-230

C:\Users\Me\Desktop>mkdir scripts
C:\Users\Me\Desktop>mkdir data
C:\Users\Me\Desktop\cds-230>dir
 Directory of C:\Users\Me\Desktop\cds-230

01/10/2013  07:25 PM    <DIR>          .
01/10/2013  07:25 PM    <DIR>          ..
01/10/2013  07:25 PM    <DIR>          data
```

```
01/10/2013  07:24 PM    <DIR>          scripts

C:\Users\Me\Desktop\cds-230>cd scripts

C:\Users\Me\Desktop\cds-230\homework2>dir
 Directory of C:\Users\Me\Desktop\cds-230\scripts

01/10/2013  07:24 PM    <DIR>          .
01/10/2013  07:24 PM    <DIR>          ..
01/09/2013  09:26 PM             13   helloworld.py

C:\Users\Me\Desktop\cds-230\scripts>python helloworld.py
Hello world!
```

Appendix B

Computational Problem Solving

Computational problem solving does not simply involve the act of computer programming. It is a process, with programming being only one of the steps. Before a program is written, a design for the program must be developed. And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested. These steps are outlined below.

```
ANALYSIS
                   Clearly understand the problem
                   Know what constitutes a solution
```

```
DESIGN
                   Determine what type of data is needed
                   Determine how the data is to be structured
                   Find another design appropriate algorithm
```

```
IMPLEMENTATION
                   Represent data within programming language
                   Implement algorithms in programming language
```

```
TESTING
                   Test the program on a selected set of problem instances
                   Correct and understand the causes of any errors found
```

1. ANALYSIS

   (a) Understanding the problem. Once a problem is clearly understood, the fundamental computational issues for solving it can be determined.

   (b) Knowing what constitutes a solution. For some problems, there is only one solution. For others, there may be a number (or infinite number) of solutions. Thus, a program may be stated as finding

- A solution
- An approximate solution
- A best solution
- All solutions

2. DESIGN

   (a) Describing the data needed. This, of course, depends on the problem at hand. We can use a list, a table, a matrix, etc.

   (b) Describing the Needed Algorithms. For some problems, there is only one solution. When solving a computational problem, either suitable existing algorithms may be found or new algorithms must be developed. Algorithms that work well in general but are not guaranteed to give the correct result for each specific problem are called *heuristic algorithms.*

3. IMPLEMENTATION Design decisions provide general details of the data representation and the algorithmic approaches for solving a problem. The details, however, do not specify which programming language to use, or how to implement the program. That is a decision for the implementation phase. Since we are programming in Python, the implementation needs to be expressed in a syntactically correct and appropriate way, using the instructions and features available in Python.

4. TESTING Software testing is a crucial part of software development. Testing is done incrementally as a program is being developed, when the program is complete, and when the program needs to be updated.

Appendix C

References

Tutorials

Tutorials for beginners:
`https://www.w3schools.com/PYTHON/python_lists.asp`
`https://www.tutorialspoint.com/python/`

A Python tutorial from the official Python website:
`https://docs.python.org/3/tutorial//`

For exact syntax and semantics of the Python language:
`http://docs.python.org/3/`

Reference manual of the standard library:
`http://devdocs.io/python`

Online Tools

The Python Online Tutor allows you to visualize execution of Python code.
`http://people.csail.mit.edu/pgbovine/python/tutor.html`

Online Python interpreter: Just in case Jupyter Notebook is not enough.
`https://www.onlinegdb.com/online_python_interpreter`

Modeling and Simulation

This book has material similar in spirit to CDS230 but slightly different approach:
`http://greenteapress.com/wp/modsimpy`