Contents

CHAPTER 1

Beginning With Python

1.1. Context

You have probablry used computers to do all sorts of useful and interesting things. In each application, the computer responds in different ways to your input, from the keyboard, mouse or a file. Still the underlying operations are determined by the design of the program you are given. In this set of tutorials you will learn to write your own computer programs, so you can give the computer instructions to react in the way *you* want.

1.1.1. Low-Level and High-Level Computer Operations. First let us place Python programming in the context of the computer hardware. At the most fundamental level in the computer there are instructions built into the hardware. These are very simple instructions, peculiar to the hardware of your particular type of computer. The instructions are designed to be simple for the hardware to execute, not for humans to follow. The earliest programming was done with such instructions. If was difficult and error-prone. A major advance was the development of higher-level languages and translators for them. Higher-level languages allow computer programmers to write instructions in a format that is easier for humans to understand. For example

```
z = x+y
```

is an instruction in many high-level languages that means something like:

(1) Access the value stored at a location labeled x
(2) Calculate the sum of this value and the value stored at a location labeled y
(3) Store the result in a location labeled z.

No computer understands the high-level instruction directly; it is not in machine language. A special program must first translate instructions like this one into machine language. This one high-level instruction might be translated into a sequence of three machine language instructions corresponding to the three step description above:

```
0000010010000001
0000000010000010
0000010110000011
```

Obviously high-level languages were a great advance in clarity!

If you follow a broad introduction to computing, you will learn more about the layers that connect low-level digital computer circuits to high-level languages.

1.1.2. Why Python. There are many high-level languages. The language you will be learning is Python. Python is one of the easiest languages to learn and use, while at the same time being very powerful: It is used by many of the most highly productive professional programmers. A few of the places that use Python extensively are Google, the New York Stock Exchange, Industrial Light and Magic, ... Also Python is a free language! If you have your own computer, you can download it from the Internet....

1.1.3. Obtaining Python for Your Computer. If you are not sure whether your computer already has Python, continue to Section 1.2.2, and give it a try. If it works, you are all set.

If you do need a copy of Python, go to the Downloads page linked to `http://www.python.org`. Be careful to choose the version for your operating system and hardware. Chosse a stable version, 3.1 or later. Do not choose a version 2.X, which is incompatible. (Version 2.6 is described in an older version of this tutorial.)

Windows	You just need to execute the installer, and interact enough to agree to all the default choices. Python works in Windows as well as on Apples and in the free operating system Linux.
OS X	Double-click on the installer. Find and run the MacPython.mpkg that is inside. Follow the defaults for installation.
Linux	Python is generally installed, though Idle is not always installed. Look for something like 'idle-python' (the name in the Ubuntu distribution).

1.1.4. Philosophy and Implementation of the Hands-On Python Tutorials. Although Python is a high-level language, it is *not* English or some other natural human language. The Python translator does not understand "add the numbers two and three". Python is a formal language with its own specific rules and formats, which these tutorials will introduce gradually, at a pace intended for a beginner. These tutorials are also appropriate for beginners because they gradually introduce fundamental logical programming skills. Learning these skills will allow you to much more easily program in other languages besides Python. Some of the skills you will learn are

- breaking down problems into manageable parts
- building up creative solutions
- making sure the solutions are clear for humans
- making sure the solutions also work correctly on the computer.

Guiding Principals for the Hands-on Python Tutorials:

- The best way to learn is by active participation. Information is principally introduced in small quantities, where your active participation, experiencing Python, is assumed. In many place you will only be able to see what Python does by doing it yourself (in a hands-on fashion). The tutorial will often not show. Among the most common and important words in the tutorial are "Try this:"
- Other requests are for more creative responses. Sometimes there are Hints, which end up as hyperlinks in the web page version, and footnote references in the pdf version. Both formats should encourage you to think actively about your response first before looking up the hint.

  The tutorials also provide labeled exercises, for further practice, without immediate answers provided. The exercises are labeled at three levels
  ***:** Immediate reinforcement of basic ideas – preferably do on your first pass.
  ****:** Important and more substantial – be sure you can end up doing these.
  *****:** Most creative
- Information is introduced in an order that gives you what you need as soon as possible. The information is presented in context. Complexity and intricacy that is not immediately needed is delayed until later, when you are more experienced.
- In many places there are complications that *are* important in the beginning, because there is a *common* error caused by a slight misuse of the current topic. If such a common error is likely to make no sense and slow you down, more information is given to allow you to head off or easily react to such an error.

Although this approach is an effective way to introduce material, it is not so good for reference. Referencing is addressed in several ways:

- An extensive Table of Contents
- Easy jumping to chosen text in a browser like Firefox
- Cross references to sections that elaborate on an introductory section
- Concise chapter summaries, grouping logically related items, even if that does not match the order of introduction.

Some people learn better visually and verbally from the very beginning. Some parts of the tutorial will also have links to corresponding flash video segments. Many people will find reading faster and more effective, but the video segments may be particularly useful where a computer interface can be not only explained but actually demonstrated. The links to such segments will be labeled. They will need a broadband link or a CD (not yet generated).

In the Firefox browser, the incremental find is excellent, and particularly useful with the single web page version of the tutorials. (It only fails to search footnotes.) It is particularly easy to jump through different sections in a form like 1.2.4.

1.2. The Python Interpreter and Idle, Part I

1.2.1. Your Python Folder and Python Examples.

First you need to set up a location to store your work and the example programs from this tutorial. If you are on a Windows computer, follow just *one* of the three choices below to find an appropriate place to download the example archive examples.zip, and then follow the later instructions to unzip the archive.

Your Own Computer: If you are at your own computer, you can put the folder for your Python programs most anywhere you like. For Chapter 4, it will be important that none of the directories leading down to your Python folder contain any blanks in them. In particular in Windows, "My Documents" is a bad location. In Windows you can create a directory in C: drive, like C:\myPython. You should have installed Python to continue.

Your Flash Drive: If you do not have your own computer, or you want to have your materials easily travel back and forth between the lab and home, you will need a flash drive.

Plug your flash drive into the computer USB port.

On the computers in the Loyola lab DH 342, you can attach to the end of a cable that reaches close to the keyboard. In DH 339, there are USB ports on the monitor. Please Note: Flash drives are easy for me to forget and leave in the computer. I have lost a few this way. If you are as forgetful as I, you might consider a string from the flash drive to something you will not forget to take with you.

Open *My Computer* (on the desktop) to see where the flash drive is mounted, and open that drive.

Temporary: If you (temporarily) do not have a flash drive and you are at a Loyola lab computer: Open *My Computer* from the desktop, and then select drive *D:*. Create a folder on drive D: with your name or initials to make it easy for you to save and remove things. Change to that folder. You should place the examples archive here. You will need to save your work somehow before you log off of the computer. You may want to email individual files to yourself, or rezip the examples folder and send just the one archive file to yourself each time until you remember a flash drive!

In Windows, after you have chosen a location for the archive, examples.zip, download it by *right* clicking on `http://cs.luc.edu/anh/python/hands-on/3.0/examples.zip` and selecting "Save As" or the equivalent on your browser and then navigate to save the archive to the chosen location on your computer. Note the the examples, like this version of the tutorial, are for Python 3.1. There were major changes to Python in version 3.0, making it incompatible with earlier versions.

If you are using Python version 2.5 or 2.6, you should continue with the older version of the tutorial. Go to `http://cs.luc.edu/~anh/python/hands-on` and find the links to the proper version of the tutorial and examples.

Once you have the archive, open a file browser window for that directory, right click on examples.zip, select Extract All. This will create the folder `examples`. End up with a file browser window showing the contents of the examples folder. This will be your *Python folder* in later discussion.

Caution 1: On Windows, files in a zip archive can be viewed while they are still in the zip archive. Modifying and adding files is not so transparent. Be sure that you unzip the archive and work from the regular directory that holds the resulting unzipped files.

Caution 2: Make sure that all the directories leading down to your Python examples directory do not include any *spaces* in them. This will be important in Chapter 4 for the local webserver. In particular, that means you should *not* place your folder under "My Documents". A directory like C:\hands-on or C:\python would be fine.

You also have the option of downloading

- An archive containing the web version of the tutorial `http://cs.luc.edu/anh/python/hands-on/3.0/handsonHtml.zip` for local viewing, without the Internet. Download it and unzip as with the

examples. The local file to open in your browser in in handsonHtml folder you unzipped and the main web page file to open is called handson.html.
- The PDF version of the tutorial for printing `http://cs.luc.edu/anh/python/hands-on/3.0/handson.pdf`.

The disadvantage of a local copy is that the tutorial may be updated online after you get your download. The change log file `http://www.cs.luc.edu/~anh/python/hands-on/changelog.html` will show when the latest update was made and a summary of any major changes.

1.2.2. Running A Sample Program.

This section assumes Python, version at least 3.1, is already on your computer. Windows does not come with Python. (To load Python see Section 1.1.2) On a Mac or Linux computer enough of Python comes installed to be able to run the sample program.

If you are in a Windows lab with Python 3.1 installed, but not set up as the default version, see the footnote.[1]

Before getting to the individual details of Python, you will run a simple text-based sample program. Find `madlib.py` in your Python folder (Section 1.2.1).

Options for running the program:

- In Windows, you can display your folder contents, and double click on madlib.py to start the program.
- In Linux or on a Mac you can open a terminal window, change into your python directory, and enter the command

  `python madlib.py`

The latter approach only works in a Windows command window if your operating system execution path is set up to find Python.

In whatever manner you start the program, run it, responding to the prompts on the screen. Be sure to press the enter key at the end of each response requested from you.

Try the program a second time and make different responses.

1.2.3. A Sample Program, Explained. If you want to get right to the detailed explanations of writing your own Python, you can *skip to the next section* 1.2.4. If you would like an overview of a working program, even if all the explanations do not make total sense yet, read on.

Here is the text of the madlib.py program, followed by line-by-line brief explanations. Do not worry if you not totally understand the explanations! Try to get the gist now and the details later. The numbers on the right are not part of the program file. They are added for reference in the comments below.

```
"""                                                    1
String Substitution for a Mad Lib                      2
Adapted from code by Kirby Urner                       3
"""                                                    4
                                                       5
storyFormat = """                                      6
Once upon a time, deep in an ancient jungle,           7
there lived a {animal}.  This {animal}                 8
liked to eat {food}, but the jungle had                9
very little {food} to offer.  One day, an             10
explorer found the {animal} and discovered            11
it liked {food}.  The explorer took the               12
{animal} back to {city}, where it could               13
eat as much {food} as it wanted.  However,            14
the {animal} became homesick, so the                  15
```

[1] If an earlier version of Python is the default in your lab (for instance Python 2,6), you can open the examples folder and double-click on the program `default31.cmd`. This will make Python 3.1 be the default version until you log out or reboot. This is only actually important when you run a Python program directly from a Windows folder. You will shortly see how to start a program from inside the Idle interactive environment, and as long as you run all your programs inside that environment, the system default version is not important.

```
explorer brought it back to the jungle,                16
leaving a large supply of {food}.                      17
                                                       18
The End                                                19
"""                                                    20
                                                       21
def tellStory():                                       22
    userPicks = dict()                                 23
    addPick('animal', userPicks)                       24
    addPick('food', userPicks)                         25
    addPick('city', userPicks)                         26
    story = storyFormat.format(**userPicks)            27
    print(story)                                       28
                                                       29
def addPick(cue, dictionary):                          30
    '''Prompt for a user response using the cue string, 31
    and place the cue-response pair in the dictionary.  32
    '''                                                33
    prompt = 'Enter an example for ' + cue + ': '      34
    response = input(prompt)                           35
    dictionary[cue] = response                         36
                                                       37
tellStory()                                            38
input("Press Enter to end the program.")               39
```

Line By Line Explanation

```
"""                                                     1
String Substitution for a Mad Lib                       2
Adapted from code by Kirby Urner                        3
"""                                                     4
```

1-4 There is multi-line text enclosed in triple quotes. Quoted text is called a *string*. A string at the very beginning of a file like this is *documentation* for the file.

5,21,29,37 Blank lines are included for human readability to separate logical parts. The computer ignores the blank lines.

```
storyFormat = """                                       6
Once upon a time, deep in an ancient jungle,            7
there lived a {animal}.  This {animal}                  8
liked to eat {food}, but the jungle had                 9
very little {food} to offer.  One day, an              10
explorer found the {animal} and discovered             11
it liked {food}.  The explorer took the                12
{animal} back to {city}, where it could                13
eat as much {food} as it wanted.  However,             14
the {animal} became homesick, so the                   15
explorer brought it back to the jungle,                16
leaving a large supply of {food}.                      17
                                                       18
The End                                                19
"""                                                    20
```

6 The equal sign tells the computer that this is an *assignment statement*. The computer will now associate the value of the expression between the triple quotes, a multi-line *string*, with the name on the left, `storyFormat`.

7-20 These lines contain the body of the string and the ending triple quotes. This `storyFormat` string contains some special symbols making it a *format string*, unlike the string in lines 1-4. The

`storyFormat` string will be used later to provide a format into which substitutions are made. The parts of the string enclosed in braces are places a substitute string will be inserted later. The substituted string will come from a custom *dictionary* that will contain the user's definitions of these words. The words in the braces: {animal}, {food}, {city}, indicate that `"animal"`, `"food"`, and `"city"` are words in a dictionary. This custom dictionary will be created in the program and contain the user's definitions of these words. These user's definitions will be substituted later in the *format string* where each `{...}` is currently.

```
def tellStory():                                          22
    userPicks = dict()                                    23
    addPick('animal', userPicks)                          24
    addPick('food', userPicks)                            25
    addPick('city', userPicks)                            26
    story = storyFormat.format(**userPicks)               27
    print(story)                                          28
```

22 *def* is short for *def*inition. This line is the heading of a *def*inition, which makes the name `tellStory` becomes *def*ined as a short way to refer to the sequence of statements that start indented on line 23, and continue through line 27.

23 The equal sign tells the computer that this is another assignment statement. The computer will now associate the name ***`userPicks`*** with a new empty dictionary created by the Python code `dict()`.

24-26 `addPick` is the name for a sequence of instructions defined on lines 29-31 for adding another definition to a dictionary, based on the user's input. The result of these three lines is to add definitions for each of the three words 'animal', 'food', and 'city' to the dictionary called userPicks.

27 Assign the name `story` to a string formed by substituting into `storyFormat` using definitions from the dictionary userPicks, to give the user's customized story.

28 This is where all the work becomes visible: Print the ***`story`*** string to the screen.

```
def addPick(cue, dictionary):                             30
    '''Prompt for a user response using the cue string,   31
    and place the cue-response pair in the dictionary.    32
    '''                                                   33
    prompt = 'Enter an example for ' + cue + ': '         34
    response = input(prompt)                              35
    dictionary[cue] = response                            36
```

30 This line is the heading of a definition, which gives the name ***`addPick`*** as a short way to refer to the sequence of statements indented on line 34-36. The name ***`addPick`*** is followed by two words in parenthesis, ***`cue`*** and ***`dictionary`***. These two words are associated with an actual cue word and dictionary given when this definition is invoked in lines 24-26.

31-33 A documentation comment for the `addPick` definition.

34 The plus sign here is used to concatenate parts of the string assigned to the name `prompt`. The current value of `cue` is placed into the string.

35 The right-hand-side of this equal sign causes an interaction with the user. The prompt string is printed to the computer screen, and the computer waits for the user to enter a line of text. That line of text then becomes a string inside the program. This string is assigned to the name `response`.

36 The left-hand-side of the equal sign is a reference to the definition of the cue word in the dictionary. The whole line ends up making the definition of the current cue word become the response typed by the user.

```
tellStory()                                               38
input("Press Enter to end the program.")                  39
```

38 The definition of `tellStory` above does not make the computer do anything besides *remember* what the instruction `tellStory` means. It is only in this line, with the name, `tellStory`, followed by parentheses, that the whole sequence of remembered instructions are actually carried out.

39 This line is only here to accommodate running the program in Windows by double clicking on its file icon. Without this line, the story would be displayed and then the program would end, and Windows would make it immediately disappear from the screen! This line forces the program to continue being displayed until there is another response from the user, and meanwhile the user may look at the output from `tellStory`.

1.2.4. Starting Idle.

The program that translates Python instructions and then executes them is the *Python interpreter*.

This interpreter is embedded in a number of larger programs that make it particularly easy to develop Python programs. Such a programming environment is *Idle*, and it is a part of the standard distribution of Python.

Read the section that follows for your operating system:

Windows (Assuming you already have Python installed.) Display your Python folder. You should see icon for `Idle31Shortcut` (and maybe a similar icon with a number larger than 31 - ignore any other unless you know you are using that version of Python). Double click on the appropriate shortcut, and an Idle window should appear. After this the instructions are the same in any operating environment. It is *important* to start Idle through these in several circumstances. *It is best if it you make it a habit to use this shortcut.* For example the alternative of opening an existing Python program in Windows XP or Vista from Open With Idle in the context menu looks like it works at first but then fails *miserably* but *inexplicably* when you try to run a graphics program.

Mac OS X the new version of Python and Idle should be in a folder called MacPython 3.1, inside the Applications folder. It is best if you can open a terminal window, change into your Python folder from Section 1.2.1, and enter the command

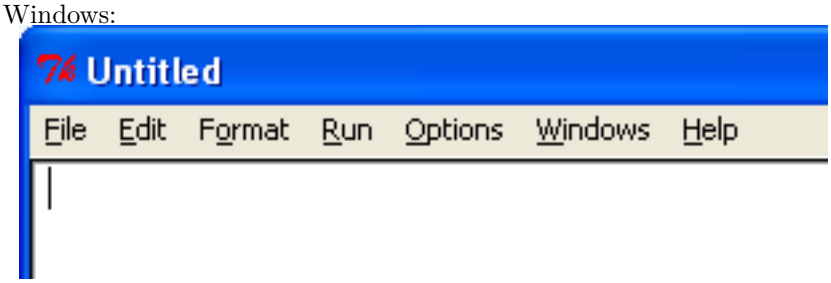
```
idle
```

If the command is not recognized, you may need to include the full path to the idle program.

Linux The approach depends on the installation. In Ubuntu, you should fine idle in the Programming section of the Applications menu. As with OS X above, you are better starting idle from a terminal, with the current directory being your Python folder.

1.2.5. Windows in Idle.

Idle has several parts you may choose to display, each with its own window. Depending on the configuration, Idle can start up showing either of two windows, an Edit Window or a Python Shell Window. You are likely to first see an Edit window, whose top left corner looks something like in Windows:



For more on the Edit Window, see Section1.9.

If you see this Edit Window with its Run menu on top, go to the Run menu and choose PYTHON SHELL to open a Python Shell Window for now. Then you may close the Edit Window.

Either initially, or after explicitly opening it, you should now see the Python Shell window, with a menu like the following, though the text may be slightly different: