## DEFINING NEW COMMANDS

In the last chapter, we explored one of the most useful features of the Python programming language: the use of the interpreter in interactive mode to do on-the-fly programming. You typed in a instruction at a prompt and instantly witnessed the computer interpret and execute the task. Although dynamic, this way of programming can be very tedious and frustrating: if you have a series of instructions that need to be followed and you make one mistake, change your mind, or need to repeat the same task multiple times, one-the-fly interpretation involves a whole lot of typing. And it is usually the same thing, over and over again. Luckily, Python is capable of more than line-by-line interactive programming, we can package multiple commands together to form a new command, called a **function**. For example, if you wanted to plot and label the line graph of the Marijuana data through time, you could define a new command `plotDrugData` as follows[1]:

```
>>> def plotDrugData():
        plot(Year, Marijuana)
        show()
        xlabel('Year')
        ylabel('Percentage')
        title('Marijuana Drug use by High School Seniors 1979-2007')
```

The first line **defines** the name of the new command to be `plotDrugData`. The lines that follow are indented slightly and contain the commands that make up the `plotDrugData` instruction, i.e., plot the data and label the plot. The indentation is an important part of Python syntax and indicates that the indented commands are part of the definition.

Once you define a new command, you can try it out by entering the command in IDLE:

```
>>> plotDrugData()
```

**Do This**: If you already have Pylab up and running, try out the new command defined above in IDLE (do not forget to import the DrugData.py file!) by entering the definition above. You will notice that as soon as you type and enter the first line, IDLE automatically indents the subsequent lines and the prompt indicator (`>>>`) disappears. After entering the last line, hit the `RETURN` key again to end the definition. Your prompt should return indicate that the command has been defined.

Run the `plotDrugData` command. Close the figure and run it again.

In Python defining and running a new command is a two step process. First, you define the function by using the **`def`** syntax as show above. Note, however, that when the function is defined, the commands that make up the function do not get carried out. Instead, you have to explicitly tell Python to issue the new command. This is called

[1] NOTE: New Python and Pylab commands will frequently be introduced throughout the text. All new commands will be summarized at the end of the chapter in which they are introduced (Python or Pylab Review sections)

***invocation***. When a function is invoked, all the commands that make up its definition will be executed in the sequence in which they are listed. Invocation allows you to use the function over and over again.

What if we wanted decided that we didn't want to plot the Marijuana data, but we wanted to plot the Cigarette data instead? We could redefine the function as follows:

```
>>> def plotDrugData():
        plot(Year, Cigarette)
        show()
        xlabel('Year')
        ylabel('Percentage')
        title('Cigarette Drug use by High School Seniors 1979-2007')
```

but doing so sort of defeats the purpose of writing a function in the first place, which is, in part, to reduce the amount of typing that we need to do.

**Adding Flexibility to Commands: Parameters**

If you look at the definition of the `plotDrugData` function, you will notice the use of parentheses, `()`, both when defining the function and invoking it. You have also used other commands, or functions, all of which have parentheses in them. Functions can specify certain ***parameters*** (or values) that impact the way instructions are carried out by placing them in parentheses. For example, the `plot` command takes two values that you specify, which indicated what data are plotted along the x- and y-axes. It can also take additional values that indicate the format of the plotted values (lines, markers, etc.):

```
>>> plot(x, y, '-rs')
```

In the above example, `x, y, and '-rs'`, are all values passed to the function through parameters. Similarly, we could have chosen to specify which drug data we want to plot by adding a parameter to the `plotDrugData` function:

```
>>> def plotDrugData(data, name):
        plot(Year, data)
        show()
        xlabel('Year')
        ylabel('Percentage')
        title(name + ' Drug use by High School Seniors 1979-2007')
```

where the first parameter is the drug as stored by Python and the second is a text string that contains the name of the drug (so that the title of the plot is correct). A function can have any number of parameters and thus can be flexible and customized. With the new definition of plotDrugData, we can now plot and label any of the data provided in the `DrugData.py` file:

```
>>> plotDrugData(Marijuana, 'Marijuana')
```

or

```
>>> plotDrugData(Heroin, 'Heroin')
```

### Saving New Commands in Modules

While a step in the right direction, this way of defining new commands is still interactive, and the result is only temporary; when you exit IDLE the function disappears and you will have to retype it the next time. Or, if you want to make a simple change (e.g., add parameters), you need to retype the whole thing from scratch. Plus, you could imagine that functions can get quite complicated (and involve a lot of steps). Luckily, Python enables you to define new functions and store them in files on your computer. Each file is called a ***module*** and can be used over and over again. Let's illustrate this by creating a module for out `plotDrugData` function:

```
# File: plotDrugData.py
# Purpose: A useful function that plots drug use through time
# Author: Your Name Here

# Import Pylab
from pylab import *

# Import Data
from DrugData import *

# Define the functions
def plotDrugData(data, name):
    plot(Year, data)    # generate the data plot
    show()              # display the figure
    xlabel('Year')      # label the plot
    ylabel('Percentage')
    title(name + ' Drug use by High School Seniors 1979-2007')
```

All lines beginning with a "**#**" sign are called ***comments***. These are annotations that help us understand and document our Python programs. Comments can be placed anywhere, including directly after a command. The `#` sign marks the beginning of a comment, which then continues to the end of the line. Anything following the `#` sign is ignored and not interpreted as a command by the computer. It is good programming practice to make liberal use of comments in all your programs so that you (and others) can read and understand your code.

Notice that we have added the `import` commands at top. If you do this for any module that requires the use of the Pylab interface, you will only have to import Pylab at the IDLE prompt if you plan to use IDLE interactively. Note: you will not always need the `DrugData.py` file, only import it when you use that data.

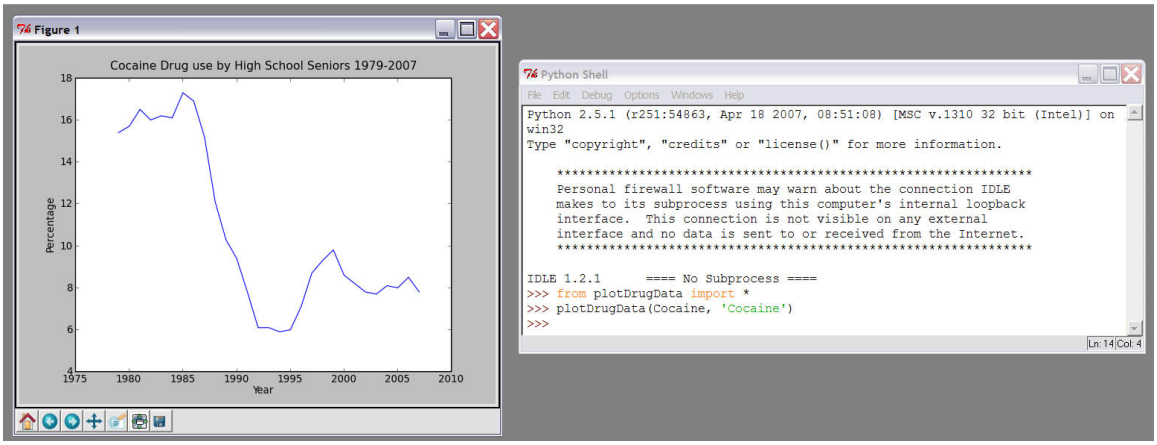
The import commands are then followed by our function definition, typed exactly as it was entered (plus some commenting) in the interactive shell. This module only contains one function definition, but modules may actually contain multiple definitions.

**Do This**: Create and store the `plotDrugData` function as a module in IDLE by first asking IDLE to open a new window (choose `New Window` from the `File` menu). Next, enter the text containing the definition above and save them in a file (let's call it `plotDrugData.py`) in your Python folder (the same place where you have your IDLE short-cut and `StartPython.py` file). You must enter the **`.py`** extension when saving your file and make sure that they are always saved in the same folder as the Python short-cut. This will make it easier for IDLE (and you) to locate your modules.

Once the file is created, you can now use it in IDLE by entering the command:

```
>>> from plotDrugData import *
```

We can then invoke the `plotDrugData` command. For example, the following plots the Cocaine data:




As you can see, accessing the command you defined is done in the same way that you access the commands in the Pylab interface. This is one of the nice features of Python, which encourages its users to extend the capabilities of the system through defining new functions and storing them in modules. Thus, importing functionality from a module you write is no different from importing commands from a module someone else has written (such as Pylab). In general the Python import command specifies two features: the module name and what is being imported:

```
from <MODULE NAME> import <SOMETHING>
```

where `<MODULE NAME>` is the name of the module (e.g., `pylab`, `plotDrugData`) and `<SOMETHING>` specifies the commands or capabilities that you are importing. By specifying `*` for `<SOMETHING>` we tell Python to import everything defined in the module. Thus, the command `from pylab import *` tells Python to import everything defined in the pylab module. Everything defined in the module is listed and documented on the Matplotlib website (http://matplotlib.sourceforge.net/backends.html; Pylab is a module that provides an interface allowing easy use the computational power of Matplotlib, an extensive math and plotting library.

**Do This**: Play around with the `plotDrugData` command for a while. Use it to plot different types of data. Modify the module so that the line that is plotted is red. Save your changes. Invoke the `plotDrugData` command again. Did the line plot in red? No it did not!

Changes made to modules are not immediately available in the interactive Python interpreter, or shell. When you modify a module, you need to recompile it so that IDLE knows that the contents have changed. Unfortunately, doing so is not as simple as re-importing the module using the import statement. Go ahead and try it. Enter the following at the prompt:

```
>>> from plotDrugData import *
>>> plotDrugData(Cocaine, 'Cocaine')
```

Did the line plot in red? No it did not!

If you were paying attention, you noticed that IDLE did not spend as much time importing the module as it had the first time you imported it during this session. That is because it recognized that the module was already loaded and did not actually re-import it! This is one defect in the IDLE interface; we need to clear its memory before we can load our modified module.

If we were not using Pylab and had started IDLE by double-clicking on the `IDLE (Python GUI)` short-cut, we would do this by selecting `Restart Shell` from the `Shell` menu. However, you've probably noticed that in your IDLE window there is no `Shell` menu! This is a consequence of the way we load IDLE so that the Pylab interface will work and it means that, unfortunately, to get IDLE to re-import a module you must *exit and then reload the shell.*

## Returning a Value

Parameters provide a way for you (or the user) to give a command information it needs to perform its task. These parameters are the ***input*** to the function. Sometimes, however, we also require a function to return some information, i.e., provide ***output***! This task involves some concepts we will be addressing in more detail later, but the actual task of returning a value is actually quite simple.

A common situation in which you might require a program or function to return a value is the performance of a simple calculation. Consider for example, the function `mean(<DATA>)` which calculates the average of the data stored in some ***variable***, indicated by `<DATA>`. Variables will be discussed in more detail in the next chapter. Executing the following in IDLE, for example:

```
>>> mean(AnyDrug)
```

will produce the following:

```
>>>
>>> mean(AnyDrug)
53.675862068965515
>>>
```

In other words, executing the command produces an output. In the above example, that output is displayed on the screen (in blue). We can also tell Python to store the value that is generated by assigning it to a *variable* through the use of the following syntax

```
>>> x = mean(AnyDrug)
```

When defining our own commands, we can let Python know that we wish the function to generate output by using the command **`return`**. For example, perhaps we get confused by the term "mean" and wish to use the term "average" instead:

```
def average(x):
    # returns the mean of the data, x
    return mean(x)
```

or alternatively

```
def average(x):
    # returns the mean of the data, x
    a = mean(x)
    return a
```

The general form of the return statement is

```
return <EXPRESSION>
```

That is, the function in which this statement occurs will return the value of `<EXPRESSION>` (and incidentally exit the function) when it encounters the return statement. Thus, the function will exit as soon as the return statement is evaluated. For example:

```
def average(x):
    # returns the mean of the data, x
    a = mean(x)
    return a
    plot(x)
```

In the above function, the final command (`plot(x)`) will not get executed! In fact, if you enter this function interactively in IDLE or in the IDLE editor, it will stop indenting, indicating that the function definition has ended.

**Functions as Building Blocks**

Now that we have learned to define new commands from existing ones, we can start to discuss a little bit more of about how Python works. The basic syntax for defining a function in Python is:

```
def <FUNCTION NAME> (<PARAMETERS>):
      <SOMETHING>
            :
      <SOMETHING>
```

In other words, to define a new function we start by using the word `def`, followed by the name of the function (`<FUNCTION NAME>`). This is then followed by any `<PARAMETERS>` enclosed in parentheses, followed by a colon (`:`). This line is followed by the commands that make up the function definition. Each command needs to be place on a separate line and should be indented the same amount. The number of spaces that makes up the indentation is not important as long as all the lines are aligned. This serves two purposes. First, it makes the definition more readable. For example, consider the following:

```
def plotDrugData():
    plot(Year, Marijuana)
      show()
    xlabel('Year')
            ylabel('Percentage')
  title('Marijuana Drug use by High School Seniors 1979-2007')

def plotDrugData():
    plot(Year, Marijuana); show()
    xlabel('Year'); ylabel('Percentage')
    title('Marijuana Drug use by High School Seniors 1979-2007')
```

The first definition will not be accepted by Python:

```
>>>
>>> def plotDrugData():
    plot(Year, Marijuana)
        show()
    xlabel('Year')
                ylabel('Percentage')
  title('Marijuana Drug use by High School Seniors 1979-2007')

  File "<pyshell#113>", line 3
    show()
    ^
IndentationError: unexpected indent
>>>
>>>
```

IDLE reports that there is an IndentationError (a type of syntax error) and either highlights or reports the line number on which the error occurred. This is because Python strictly enforces the indentation rule described above. The second definition, however, is acceptable, for two reasons: 1) indentation is consistent and commands can be entered on the same line if they are separated by a semi-colon (`;`). We would recommend that you continue to enter each command on a separate line until you are more comfortable with

Python. Plus, it makes code a lot easier to read. Most importantly, however, you will notice that IDLE makes indentation easy by usually doing it for you!!

Another feature of IDLE that enhances readability of Python programs is the use of color to highlight certain types of syntax. In the screen shot above, for example, the word `def` is in orange, the name of the function is in blue, strings are in green, and the error message is in red. Other colors are used in other situations.

The idea of code re-usability that is introduced through the use of functions is a very powerful and important concept in computing. When we define a new command using existing commands we *abstract* a new set of instructions for the computer to perform. We can also define other functions that use functions we have already defined (such as `plotDrugData`). Thus, functions are the basic building-blocks of the bigger structure that is a complex program.

## PYTHON REVIEW

In this exercise we learned several Python commands. These included:

- **Importing Modules:**

  ```
  from <MODULE NAME> import <SOMETHING>
  ```

- **String Concatenation:**

  A phrase in between two single quotes (e.g., `'Any Drug'`) is a special piece of code called a **string**. We will be talking about more strings later, but in this chapter we did a bit of string manipulation that needs to be addressed. In the plotDrugData module we included the following instruction:

  ```
  title(name + ' Drug use by High School Seniors 1979-2007')
  ```

  which told Python to set the text title to the string provided in the parameter, name, added to the start of `' Drug use by High School Seniors 1979-2007'`. Try it out by entering and concatenating strings in IDLE.

- **Function Definitions:**

  ```
  def <FUNCTION NAME>(<PARAMTER>):
      <SOMETHING>
          :
      <SOMETHING>
  ```

- **Return Statements**

  ```
  return <EXPRESSION>
  ```

- **Multiple commands can be entered on the same line if separated by a semi-colon (`;`)**

- **Arrays:**

  Arrays are lists of values stored as a single variable, using the syntax:

```
myArrayName = [value1, value2, value3, ...]
```

## PYLAB REVIEW

In this exercise we learned several Pylab commands. These included:

- **Labeling Plots**

  *Axes*

```
xlabel('label for x axis')
ylabel('label for y axis')
zlabel('label for z axis')
```

  *Title*

```
title('plot title')
```

  where the phase in between the single quotes is a *string* that indicates the text to be printed. Like lines, labels have properties that can be modified using keywords. These include the following:

```
color: a matplotlib color argument
fontsize: a scalar value
fontweight: 'bold', 'normal', 'light'
fontangle: 'normal', 'italic', 'oblique'
font: 'Courier', 'Times', 'Sans', 'Helvetica'
horizontalalignment: 'left', 'right', 'center'
verticalalignment: 'bottom', 'center', 'top'
```

  for example

```
>>> xlabel('Year', color = 'r', fontweight ='bold')
```

  adds the a red, bold label, "Year" to the x-axis.

- **Calculating an Average:**

  The function `mean(<ARRAY OF DATA>)` can be used to calculate the average for a list of numeric data values.

- **Plotting**

The `plot()` command can be invoked with either one or two arguments.

```
plot(x)      # plots all values of the single variable x
plot(x, y)   # plots all values of x against values of y;
             # x and y must have the same number of data points
```

## EXERCISES

1. Read Chapters 2 and 3 from the Tufte book. Pick a figure in Chapter 2 that Tufte argues is distorted but for which a correction is not provided. Why is this figure distorted? How should the distortion be corrected? In Chapter 3, Tufte discusses why artists draw graphics that lie. Considering that most scientists are not artists, in addition to the reasons listed in the Chapter, why else would a scientific "artist" produce a graphic that lies? Are any of these reasons justified (ethically)? (10 pts)

2. Define and save as a module a function that generates a series of bar charts on a single figure that plots the following drugs versus Year: Alcohol, Cigarettes, Marijuana, and Cocaine (the order is important to be able to visualize all the data). You can set a color for each bar graph using the keyword argument `color` (e.g., `color = 'k'`). The function should also label the axes and title the graph. Run you function in IDLE and save the result. What does this visualization tell you about the relationship among the use of these drugs over time? Is this a good visualization? Why or why not? (15 pts)

3. Define and save as a module a function that plots one type of drug data versus another (e.g., Marijuana versus Cocaine use). The function should have parameters that allow you to specify the two drugs to be compared and the format of the line (e.g., solid red line `'r'`, dashed blue line `'--b'`, magenta squares `'ms'`). The function should also label the axes and title the graph. Use your function to generate a variety of comparisons; save your two favorites and explain the relationship illustrated by each visualization. (15 pts)

4. Define and save as a module a function that calculates the average drug use from 1979 to 2007 for each type of drug (except AnyDrug). You can return the calculated values by creating a type of variable called an array that stores all of the results with a single name using the following syntax:

   ```
   averageUse = [aMarijuana, aHallucinogens, aCocaine, etc...]
   ```

   and then returning the `averageUse` variable. Test your function in IDLE. (25 pts)

5. Define and save as a module a function that calculates the average drug use for each type of drug (except AnyDrug) and then plots the average drug use for each type of drug as a scatter plot using the `plot` command (which can generate univariate as well as bivariate graphs). HINT: you already did part of this!!! The function should also label the axes and title the graph. Labeling the ticks on the x-axis correctly is actually quite a difficult task and we will leave it for another day. When you print out your

final plot, you can label the ticks manually with a pencil so that it is easier to interpret the plot. Or for the computer savvy you could edit the tick labels in a graphics editor or paint program. Save and run the module; save the resulting plot and explain the illustrated relationship. What insight does this visualization give you into the relative use of each drug by high-school seniors over the past 28 years? Is this a good visualization? Why or why not? (20 pts)