

Programming for Psychology in Python

Fundamentals

0. **Introduction**
 1. [Getting started](#)
 2. [Data—numbers, variables, and functions](#)
 3. [Data—strings and booleans](#)
 4. [Data—lists](#)
 5. [Flow control—conditionals](#)
 6. [Flow control—loops](#)
 7. [Dealing with errors](#)
 8. [Exercise solutions](#)
-

Introduction

Welcome to this set of lessons on the fundamentals of programming for psychology using the Python language. The aim of these lessons is to provide you with a foundational knowledge of programming and Python, which will prepare you for subsequent lessons that use such technologies in psychology applications.

What is programming and what is Python?

A program is a set of instructions for a computer to perform a particular set of tasks—programming is the process of developing these instructions. Computer programs can be written in many different languages, from "low-level" languages where the instructions are closely tied to the computer's hardware to "high-level" languages that are more human-readable.

Here, we will be using a programming language called [Python](#)—a high-level language that is very readable and quick to get up-and-running for new users. It is one of the most popular languages for computer programming, and is currently the most frequently taught language for introductory computer science students in the USA.

Python is far from just a language that is useful for education, however. It is used to run some of the most popular websites in the world, such as YouTube, Reddit, and Google Maps. It also has extensive roots in the scientific community, with comprehensive Python frameworks available for many disciplines.

Why is programming useful to learn as a psychology student?

The main benefits of learning programming as a psychology student stem from its usefulness in conducting research and how it can develop cognitive skills:

- *It allows greater flexibility and independence in research.* Much research involves doing something that has never been done before. Knowing a programming language such as Python gives you the power to use your computer in ways that go far beyond what can be offered by any general or specialist application.
- *It promotes sound research practices.* Your code becomes an accurate record of precisely what was involved in your research. Anybody with access to your code can have a clear understanding of what it was that you did in your research.
- *It develops algorithmic and problem-solving abilities.* Successfully programming a solution to a particular task requires an algorithmic approach and a problem-solving mindset. Furthermore, as you will see, programming is very unforgiving of mistakes or imprecision—hunting down the cause of coding errors ('bugs') is in itself an excellent way of honing problem-solving skills and strategies.

How are these lessons organised?

Each lesson is focused around a key concept in the fundamentals of Python programming. The recommended approach is to follow along with the content while writing and executing the associated code.

You will also find that each lesson has an associated screencast. In each screencast, I narrate the process of coding

and executing scripts related to the concept of the lesson. The purpose of these screencasts are to give a practical demonstration of the concepts of the lesson. They will typically cover much the same content as the written material. It is recommended that you both view the screencast and read through the written material.

You can also view the written material of all the lessons as a [combined PDF](#).

You will notice that there is a later lesson called [Dealing with errors](#). If you are encountering an error that you are having trouble understanding, feel free to jump ahead to this lesson and see if you can get more information on its cause and possible remedy.

The lessons also contain the occasional *tip*, which are small concepts that give additional suggestions on the relevant content (sometimes highlighting very important concepts). An instructive example is:

Tip: It is best to watch the screencasts in HD resolution—otherwise the text is not very legible. If you don't have a fast enough network connection, you can copy the video (mp4) files from the course directory and view those rather than streaming.

Finally, the lessons also include exercises for you to try.

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Fundamentals

0. [Introduction](#)
 1. **Getting started**
 2. [Data—numbers, variables, and functions](#)
 3. [Data—strings and booleans](#)
 4. [Data—lists](#)
 5. [Flow control—conditionals](#)
 6. [Flow control—loops](#)
 7. [Dealing with errors](#)
 8. [Exercise solutions](#)
-

Getting started

We first need to understand how we can develop and execute Python scripts. We are going to use an application called [Spyder](#) as our development environment.

Objectives

- Be able to run Spyder and understand its role in executing Python scripts.
- Know the key components of the Spyder interface.

Screencast

Executing Python scripts

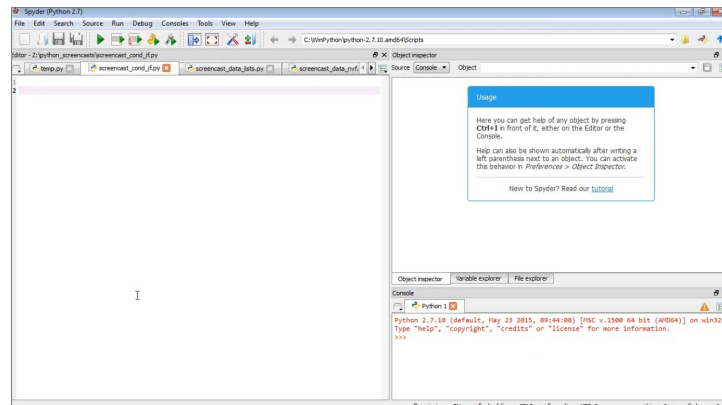
The process of developing and executing Python scripts has two different elements. First, we write our code into a text editor (similar to how you might write something in Microsoft Word), saving it as a file with a ".py" extension ("analysis_script.py", for example). Then, we pass this file name on to Python, which interprets and executes the code that is contained within the file.

In these lessons, we will be using Spyder as our development environment to simplify this process and make it

accessible within the one application.

Starting Spyder

If you are on a computer in the Mathews Building, there will be an icon on the desktop named "Spyder". Double-click on it to launch the application, which will open a window that looks like that shown below (click to enlarge):



Installing Python and Spyder on your own computer

For the purposes of this course, we will be assuming that you are using the Python and Spyder environment on the School of Psychology computers. However, all the software that we will be using is free and open-source, so it is possible for you to install and run it on your personal computer.

Windows

If you're using Windows, the easiest way is to copy the "unsw_psy_wpython_2016.zip" file on the course shared drive to a USB drive, and then extract the contents into the "C:\WinPython" directory on your computer (it is important to note that while I have no reason to believe that this software will pose a problem for your computer, you are installing the software at your own risk and I cannot take responsibility for any unexpected effects). To execute Spyder, navigate to C:\WinPython using Windows Explorer and double-click on `Spyder.exe`.

Mac

If you're using a Mac, the process is a bit more involved but the following appears to work.

1. Download "Anaconda" "Graphical Installer" from [this website](#).
2. Run the file that you downloaded, installing Anaconda into the desired location on your computer (we will assume `/Applications`).
3. Use the finder to navigate to `/Applications/anaconda` and run the Launcher.
4. Click on the section at the top that says `Environment: root` and then click on `Clone Environment "root"`. Call it something like `psych`. This command may then take a bit of time to complete.
5. Open up the Mac terminal. This can be found by either searching for "terminal", or selecting "Go" from the finder top menu and then "Utilities" and then "Terminal".
6. Inside the terminal, execute the following commands in sequence. If asked, press `y` to confirm an operation:

```
cd /Applications/anaconda/bin
source activate psych
pip install pyglet pyopengl
conda install pillow wxpython
pip install psychopy
conda install -c https://conda.anaconda.org/lidavidm --no-deps veusz
```

7. Run the Launcher again (as in Step 3). Right-click inside the Launcher and click `Reload`.
8. If all goes well, you should then be able to use the "Launcher" to start Spyder and use it as shown in the screencasts.

Once you're able to start Spyder, there are some default settings that are worth changing. Open the Preferences window, and apply the following:

- Run
 - Set "Console" to "Execute in a new dedicated Python console"

- Editor
 - "Code Introspection/Analysis" tab, disable "Automatic code completion", "Case sensitive code completion", "Enter key selects completion", and "Link to object definition".
 - "Advanced settings" tab, disable "Automatic insertion of parentheses, braces and brackets" and "Automatic insertion of colons after 'for', 'if', 'def', etc"

Linux

If you're using Linux, let me know and we can work out a way to install it.

The Spyder interface

There are four main elements to the Spyder interface to be aware of:

1. *The editor.* This is the large vertical pane on the left, with the numbered lines down the side. This is where we will type in our Python commands.
2. *The help.* This is the pane on the upper right side. We will use this region to display help with Python commands.
3. *The "run" button.* This is the green "play" icon (triangle pointing to the right) on the icon bar towards the top of the screen. We will use this to tell Python to execute our current script.
4. *The output.* This is the pane on the lower right side. We will use this to display any output from our Python scripts, and any error messages it may produce.

A first script

Let's finish by creating our first Python script—one that displays the phrase "Hello, world!" to the output window.

First, create a blank file by clicking File → New or clicking on the new file icon. Then, click File → Save As. As we proceed through the lessons, you will need to save your Python code into various files. Let's start by creating a directory in which we will save them. As a suggestion, you may like to create a directory on your Z drive. When inside the directory, give the current file a name such as "hello_world.py" and save it.

Now we are ready to create our Python code. To print "Hello, world!" to the output, we type into the editor pane:

```
print "Hello, world!"
```

We then save the file before executing it by clicking on the green arrow icon. If all goes well, this should show the following in the Spyder output window:

```
Hello, world!
```

Summary

The overall objective of this lesson was to introduce you to the Spyder application, which we will use as our environment for developing and executing Python scripts. We looked at how to start Spyder, how it fits in with the process of Python programming, the most important elements of its interface, and we used it to create and run our first script.

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Fundamentals

0. [Introduction](#)
 1. [Getting started](#)
 2. **Data—numbers, variables, and functions**
 3. [Data—strings and booleans](#)
 4. [Data—lists](#)
 5. [Flow control—conditionals](#)
 6. [Flow control—loops](#)
 7. [Dealing with errors](#)
 8. [Exercise solutions](#)
-

Data—numbers, variables, and functions

Objectives

- Be able to apply basic operations to numbers.
- Understand the role of variables and how to use them appropriately to represent data.
- Be able to understand and use functions

Screencast

Numbers

We will start our investigation of Python programming by thinking about a very simple programming environment that you are all likely to be familiar with—a calculator. If you think about the basic functionality of a calculator, it allows you to enter representations of numbers, perform operations on them, and display the output. We can approach Python in the same way. For example, if we wanted to add two numbers together and display the result, we could write the following Python code:

```
print 2 + 2
```

If we then save the file containing this code and run it, we can see that it produces the following output, as expected:

```
4
```

This example shows that Python is aware of a concept of a number and has the potential to apply mathematical operators to them (addition, in this case). Let's have a look at a few more operations:

```
# multiplication
print 2 * 3
# division
print 2 / 2
# exponentiation
print 2 ** 4
# subtraction
print 2 - 4
```

```
6
1
16
-2
```

Tip: You can see in the above several lines that begin with "#". Starting a line with the "#" character tells Python that the rest of the line is a comment, and it will not attempt to interpret it. We can put whatever we like in comments, and they are a very useful way of explaining our code using natural language.

In the above, we represented the numbers as integers. We can also represent them as decimals, such as:

```
print 2.0 ** 3.0
```

```
8.0
```

Tip: The difference between representing a number as an integer and as a decimal can be potentially important in some cases. Unless what you're representing can only be an integer, it is best to use a decimal representation.

Variables

Programming becomes much more powerful once we introduce *variables*. By using variables, we can store and refer to data during the operation of our program.

For example, say we have 3 apples and 2 oranges. We can write something like:

```
apples = 3
```

Here, we've created a variable called `apples`. We've given it an informative name, then the equals sign (to indicate *assignment*), and then the value that it will refer to (the integer 3, in this case).

Tip: We can call our variables close to whatever we want; they can't start with a number, contain any spaces, or come from a set of words that are special to Python, but in general we are free to choose our names as we please.

Now we can proceed and create another variable for our oranges:

```
apples = 3
oranges = 2
```

Now that we have specified our two variables, we can use them to refer to their values. For example:

```
apples = 3
oranges = 2
print apples + oranges
```

Functions

By using functions, we can expand the range of operations that we can apply to our data.

Before introducing functions, we will first consider another concept in Python programming—how we can make additional functionality available to our scripts. The Python language has a lot of built-in functionality available, but an immense amount of functionality is optional and we need to tell Python that we want to make it available. To do this, we use the `import` command.

For example, a functionality that is required surprisingly often in programming is the capacity to generate random numbers. Luckily, Python has such functionality—but it is an optional extra so we first need to tell Python that we want to use it:

```
import random
```

Now we have the functionality of the `random` package available to us. To generate a random number, we'll use a function called `random`:

```
import random

rand_val = random.random()

print rand_val
```

```
0.344162398107
```

Tip: What functions are available in a package? In Spyder, you can type the package name followed by a dot (e.g. `random.`) and then press the `TAB` key to get a list of the package functionality, which includes any available functions.

Functions can take values called *arguments*, which affect how the function operates. Such arguments are specified by including them in parentheses; in this example, we do not need to provide any arguments so we simply open and close our parentheses. The function `random` returns a random number between 0 and 1, which we assign to the variable `rand_val`, which we then print to the screen.

Tip: How do you know what arguments a function is expecting? The best way is to write the function name in Spyder and then press `CTRL-I` while the cursor is nearby to the function name. The help window will then show the information about the function, including its arguments. You can also look at its [help website](#).

Let's look at another example of a function, this time where we do want to specify some arguments. Say if we want to generate a random number from a Gaussian (Normal) distribution, rather than from a uniform distribution between 0 and 1. As we know, a Gaussian distribution requires two pieces of information—its mean and its standard deviation. So if we want to generate a random number from a Gaussian distribution with a mean of 0 and a standard deviation of 1, we can write:

```
import random

gauss_val = random.gauss(0, 1)

print gauss_val
```

```
0.175366331357
```

Instead of simply specifying the value of a particular argument, we can also explicitly tell the function what the value of a particular argument is. This is often good practice, as it makes the function arguments very clear. For example:

```
import random

gauss_val = random.gauss(mu=0, sigma=1)
```



```
print gauss_val
```

```
-1.24025255643
```

Of course, we do not need to specify values directly but can instead use variables:

```
import random

gauss_mean = 0.0
gauss_stdev = 1.0

gauss_val = random.gauss(mu=gauss_mean, sigma=gauss_stdev)

print gauss_val
```

```
-0.184589263803
```

Exercises

1. Investigate how the following operators work: `>`, `<`, `>=`, `<=`. We will talk more about the type of data these operators produce in the next lesson.
2. Parentheses can be used to impose an order of operation. Compare the output of:

```
2 + 3 * 5
```

and

```
(2 + 3) * 5
```

3. What does the `//` operator do? (Hint: compare the operation between integers and decimals)
4. Investigate the use of the `random.triangular` function. What arguments does it require?
5. Import the `math` package and investigate its functionality. What function might be useful to convert a value in radians to its equivalent in degrees?

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Fundamentals

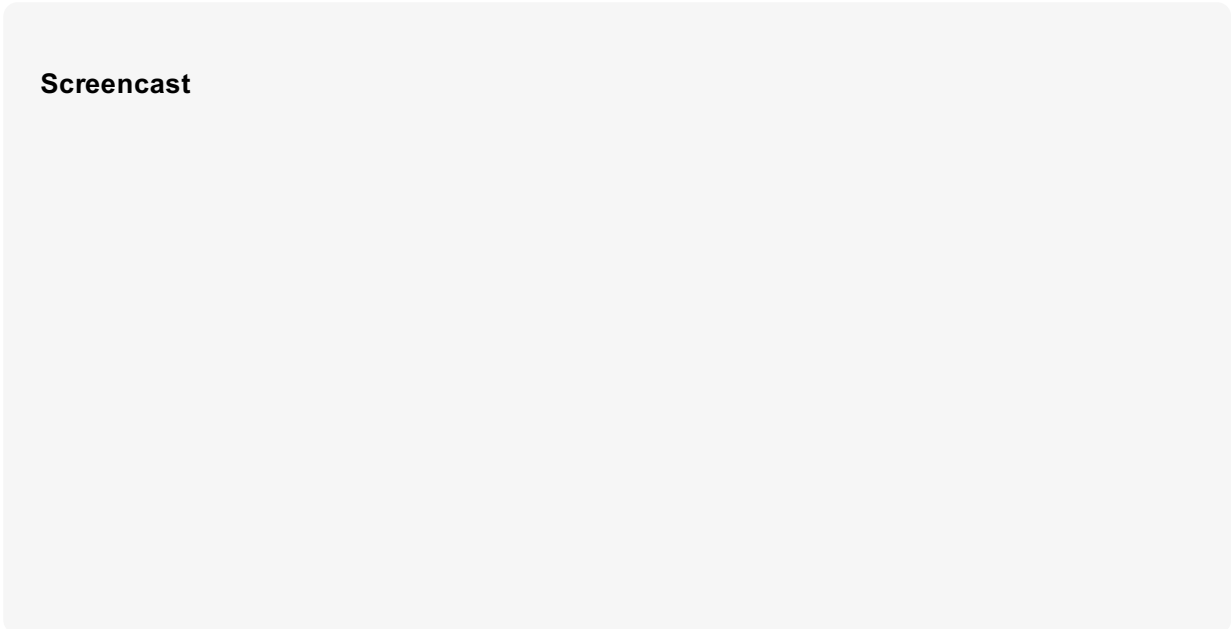
0. [Introduction](#)
 1. [Getting started](#)
 2. [Data—numbers, variables, and functions](#)
 3. **Data—strings and booleans**
 4. [Data—lists](#)
 5. [Flow control—conditionals](#)
 6. [Flow control—loops](#)
 7. [Dealing with errors](#)
 8. [Exercise solutions](#)
-

Data—strings and booleans

Objectives

- Understand the concept of strings and textual representations.
- Be able to use functions attached to data.
- Understand boolean data and the use of logical operators.

Screencast



Strings

Creating strings

We are first going to investigate strings, which are how Python represents text data. We've actually already come across strings, when we executed the following code:

```
print "Hello, world!"
```

The collection of characters that is enclosed in quotation marks (") defines a string. As we saw in the last lesson, we can assign such data as a variable:

```
hello_text = "Hello, world!"  
  
print hello_text
```

Accessing characters in strings

The variable `hello_text` refers to a string, which is a collection of characters. We can access the individual characters using square brackets and an *index*. Such indices start at 0; for example, to access the first character only we would append `[0]`, as below:

```
hello_text = "Hello, world!"  
  
print hello_text  
print hello_text[0]
```

```
Hello, world!  
H
```

We can also access a range of characters by using separate indices separated by colon (:) characters. As we've seen, the first number specifies the start index. The second specifies the end index; what is extracted is up to but not including this index. For example, if the first index is 0 and the second index is 2, the characters that are returned would be those in the 0 and 1 indices (not 2). Finally, the last number is the 'step', which defines the increment. For example, if it is 2, then we extract every second character. Putting it all together, we could extract the `Hello` component of the `Hello, World!` string by doing:

```
hello_text = "Hello, World!"  
  
print hello_text[0:5:1]
```

```
Hello
```

We can also use some shortcuts when using this form of indexing. First, if we are starting at 0 then we do not need to specify the number, as it is assumed. Second, if our 'step' is 1, then we do not need to specify either the number or the colon, as it is assumed. For example:

```
hello_text = "Hello, World!"  
  
# these are all identical  
print hello_text[0:5:1]  
print hello_text[:5:1]  
print hello_text[:5]  
print hello_text[:5]
```

```
Hello  
Hello  
Hello  
Hello
```

This method of accessing components of a variable using square brackets is an important concept, and one that we will return to later.

Using operators with strings

We can also use some of the operators we've encountered already on strings, which behave intuitively:

```
hello_text = "Hello, world!"  
  
print hello_text + " From Python"  
  
print hello_text * 2
```

```
Hello, world! From Python  
Hello, world!Hello, world!
```

Of course, some of the operators don't make very much sense in the context of strings and Python will complain if we

try to use them:

```
hello_text = "Hello, world!"  
  
print hello_text ** 2
```

```
Traceback (most recent call last):  
  File "<string>", line 3, in <module>  
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

As we can see, when Python encounters the offending line it causes an error—the program stops executing and displays an error message. The error message is somewhat informative—Python doesn't know how to combine a string and an integer in an exponentiation operation.

Tip: Being able to interpret Python's error messages is an important skill. When you encounter an error, give the associated message a close read. It will tell you the line number of the code that is causing problems, and give you some clues on the nature of the error.

An important point is that what seems like numbers are no longer considered as numbers once they are placed inside quotation marks; they acquire the properties of strings rather than of numbers. For example:

```
oranges = "2"
```

Here, the variable `oranges` does not refer to the *number* 2, but to a string that contains the character 2. What are the consequences of this? As we've seen, we can add numbers and add strings—we can see that Python treats `oranges` as a string by considering:

```
oranges = "2"  
  
print oranges + oranges
```

```
22
```

Functions attached to data

Strings are a good example of an important aspect of Python programming—using functions that are attached to data. Every string that we define comes with a set of associated functions that can be very useful. For example, the `upper` function converts all the characters in the string to upper case:

```
hello_text = "Hello, world!"  
  
print hello_text.upper()
```

```
HELLO, WORLD!
```

Tip: As we encountered before, we can investigate what functions are attached to a given variable using Spyder by typing the variable name followed by a dot and then pressing `TAB`. Combined with using `CTRL-i` to show the help for a given function, this is a powerful way to determine the functionality attached to a variable.

Boolean data

Now we will consider another important type of data in Python, *booleans*. This is a straightforward type of data that can only be one of two values: `True` or `False`.

Tip: Note the capitalisation of `True` and `False`. Python is generally case-sensitive, so `true` is not the same as `True`.

As usual, we can assign boolean data to variables. For example, say we are running an experiment where we don't want our program to use the whole of the screen and we are using the Windows operating system. To indicate this, we might define the following variables:

```
fullscreen = False
is_windows = True
```

We can combine booleans using logical operators. For example, say if we wanted to determine whether we want to run in fullscreen and we are using Windows:

```
fullscreen = False
is_windows = True
print fullscreen and is_windows
```

```
False
```

The `and` operator returns `True` only if both its inputs are `True`. On the other hand, the `or` operator returns `True` if either of its inputs are `True`:

```
fullscreen = False
is_windows = True
print fullscreen or is_windows
```

```
True
```

We can also do negation, which flips around our boolean value:

```
fullscreen = False
print fullscreen
print not fullscreen
```

```
False
True
```

Booleans are particularly useful when we want to control our program flow, which we will see in a later lesson. This often involves testing if two things are equal, which involves boolean data. For example, say we define a subject identifier somewhere in our code:

```
subj_id = "p1001"
```

At some point in our code, we might want to know if the subject ID is "p1001". We can test for this using the comparison operator `==`:

```
subj_id = "p1001"
print subj_id == "p1001"
```

```
True
```

Note the use of the double equals sign in the second line of code above. This is very important, and rather tricky. The single equals sign, `=`, on the first line of code means *assignment*. By using the double equal sign in the second line of code, `==` we are doing a *comparison*.

For example, the following *assigns* the value of 2 to the variable `oranges`.

```
oranges = 2
```

Whereas the following *compares* the value of the variable `oranges` to the value 2, giving either `True` or `False`, which is then assigned to the variable `are_there_two_oranges`.

```
oranges = 2
are_there_two_oranges = oranges == 2
```

Exercises

1. What are two ways in which you can access the *last* character in a string? Hint: investigate the `len` function and what happens when using indices that are negative numbers.
 2. Use a function attached to the string `Hello, world!` to change it to `Hello, Sydney!`.
 3. String data has a bunch of functions attached to it that begin with `is` (e.g. `isalnum`), that return boolean values. What do each of these functions test? Which one returns `True` for the string `Hello, World!`?
 4. There is also a comparison operator for non-equality: `!=`. Try using it to see if a given string is not equal to `p1001`.
 5. Does testing for equality also work on numbers?
-

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Fundamentals

0. [Introduction](#)
 1. [Getting started](#)
 2. [Data—numbers, variables, and functions](#)
 3. [Data—strings and booleans](#)
 4. **Data—lists**
 5. [Flow control—conditionals](#)
 6. [Flow control—loops](#)
 7. [Dealing with errors](#)
 8. [Exercise solutions](#)
-

Data—lists

Objectives

- Be able to define a list and use some of its keys functions.
- Know how to access list elements.

Screencast

Lists

In this lesson, we're going to investigate how we can represent a *collection* of data in Python using a data type called a *list*. Some of this will be familiar from the lesson on strings, because they share many of the same properties and features as lists.

Let's begin by creating a list. We do this by listing a series of values, separated by commas, and enclosing them within square brackets. For example:

```
apples = [2, 4, 6, 5]
```

We've created a variable named `apples` that refers to a list with 4 items, where each item is a number.

Similar to strings, lists have a set of functions attached to them that are often quite useful. For example, the `append` function can be used to add additional elements to the list. So if we wanted to add another element, the number 2, we could run:

```
apples = [2, 4, 6, 5]

print apples

apples.append(2)

print apples
```

```
[2, 4, 6, 5]
[2, 4, 6, 5, 2]
```

We can use another function, `count`, to determine how many elements in the list match a particular target value. For example:

```
apples = [2, 4, 6, 5, 2]

print apples.count(2)
print apples.count(4)
print apples.count(1)
```

```
2
1
0
```

An important property of lists is that they don't need to have elements that are all of the same type. For example, it is fine to mix strings and numbers in the same list. Lists can even contain other lists, which is an important concept. For example:

```
apples = ["a", 1, "b", 2]

print apples

apples = [["a", 1], ["b", 2]]

print apples
```

```
['a', 1, 'b', 2]
[['a', 1], ['b', 2]]
```

We can access the individual elements of a list just like we did with strings. For example:

```
apples = [["a", 1], ["b", 2]]

print apples[0]
print apples[0][0]
print apples[1][0]
```

```
['a', 1]
[['b', 2], ['a', 1]]
a
```

The last example in the above is an extension of what we've covered already, and shows how element access can be chained together. In this example, we access the first element (zeroth index) of `apples`, which is the list `["a", 1]`. We then access the first element (zeroth index) of this list, which is "a".

Finally, we'll consider some important functions that can be used to create lists. One function in particular that is often encountered and is very useful is `range`. When provided with one argument, the `range` function generates a list of numbers from 0 up to, but not including, the value of the argument. For example:

```
print range(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Similar to how we accessed multiple list elements, we can also provide the `range` function with start, end, and step

values:

```
print range(2, 10, 3)
```

```
[2, 5, 8]
```

Exercises

1. Which mathematical operators work with lists? Do they behave how you would expect?
2. Investigate the attached `remove` function of a list. What is something that you would need to be careful of when using this function?

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Fundamentals

0. [Introduction](#)
 1. [Getting started](#)
 2. [Data—numbers, variables, and functions](#)
 3. [Data—strings and booleans](#)
 4. [Data—lists](#)
 5. **Flow control—conditionals**
 6. [Flow control—loops](#)
 7. [Dealing with errors](#)
 8. [Exercise solutions](#)
-

Flow control—conditionals

Objectives

- Know how to use the `if` statement to control the path of program execution.

Screencast



Conditionals

In this lesson, we will look at how we can control the flow of our programs through the use of conditional statements. We're going to do this using the family of `if` statements in Python.

We will approach conditionals in the context of a mock experiment. Say you are running an experiment with three conditions, represented by the numbers 1, 2, and 3. On a given trial, the stimulus shown to the participant depends on the condition number for that trial. For the purposes of this demonstration, our experiment will print `blank` if the condition number is 1, `A` if the condition number is 2, and `B` if the condition number is 3.

We can implement our experimental design using conditionals. For example:

```
cond_number = 2

if cond_number == 1:
    print "blank"
```

Let's unpack this example. First, we set a condition number for this trial. Then comes the conditional aspect to the code. We start with the command `if`, which is a special Python command. We followed it by something that evaluates to a boolean; in this case, a test of equality. Then we have a colon character. You can read this line as "if the condition number is equal to 1". We then start a new line, indent by pressing the `TAB` key, and then specify a print statement.

Tip: This usage of indentation is very important to understand. Indentation is the method Python uses to indicate nested structure. It is not just for aesthetics—how the program operates is directly connected to the indentation structure of the code.

If we were to run this code, then it would produce no output. This is because the indented `print` statement is only executed if the `if` statement above it evaluates to `True`. Because the condition number is not equal to 1, the print statement is skipped and the program finishes.

To implement our experiment design, we have to include some additional conditional statements. We can do this using the `elif` (i.e. else if) command:

```
cond_number = 2

if cond_number == 1:
    print "blank"

elif cond_number == 2:
    print "A"
```

```
A
```

You can see that we've removed the indentation, because we don't want our next conditional statement to be 'under' the previous `if`. We've used the command `elif`, followed by a test of whether the condition number is equal to 2. Because this evaluates to `True`, the program descends into the indented section and executes the `print` statement.

Let's complete our experimental design by adding another `elif` statement:

```
cond_number = 2

if cond_number == 1:
    print "blank"

elif cond_number == 2:
    print "A"

elif cond_number == 3:
    print "B"
```

```
A
```

Finally, we will consider the last construct in the `if` family, the `else` command. Any statements that are nested under the `else` block are executed if none of the previous `if` or `elif` commands evaluated to `True`. In our example, the only way this could happen is if there is a problem with the condition number—so we will print a message to the user to that effect:

```
cond_number = 2

if cond_number == 1:
    print "blank"

elif cond_number == 2:
    print "A"

elif cond_number == 3:
    print "B"

else:
    print "Unexpected condition number"
```

```
A
```

Let's wrap up by going through the program logic. First, we set a condition number. Then, we test if the condition number is equal to 1. If this is true, then we print the output `blank` to the screen. Because something in the construct has evaluated to `True`, we skip the rest of the tests and the program finishes. If it evaluates to `False`, then we move down to the next `elif` statement, which asks if the condition number is 2. If that is true, then it prints `A` to the screen before moving out of the construct and finishing. If it is false, then we move down to the next `elif` statement and do the same thing for the condition number 3. Finally, if nothing has evaluated to `True`, we enter the `else` block and execute its print statement.

Let's just check that our experimental design works as expected:

```
cond_number = 1

if cond_number == 1:
    print "blank"

elif cond_number == 2:
    print "A"

elif cond_number == 3:
    print "B"

else:
    print "Unexpected condition number"
```

```
blank
```

```
cond_number = 2

if cond_number == 1:
    print "blank"

elif cond_number == 2:
    print "A"

elif cond_number == 3:
    print "B"

else:
    print "Unexpected condition number"
```

```
A
```

```
cond_number = 3

if cond_number == 1:
    print "blank"

elif cond_number == 2:
    print "A"

elif cond_number == 3:
    print "B"

else:
    print "Unexpected condition number"
```

```
B
```

```
cond_number = 300

if cond_number == 1:
    print "blank"

elif cond_number == 2:
    print "A"

elif cond_number == 3:
    print "B"

else:
    print "Unexpected condition number"
```

Exercises

1. Imagine that you have a subject ID stored as a string (e.g. `subj_id = "p1000"`). Write a script that uses a function attached to `subj_id` to print `A` if the subject ID starts with `p1` and `B` if the subject ID starts with `p2`.
2. Your task is to print `A` if the condition number is between 1 and 10. There are at least three different ways to do this—see if you can work out what they are. Which method do you prefer?
3. Will `B` be printed to the screen in the code below? Why or why not?

```
cond_number = 2
subj_id = "p1001"

if cond_number == 2:
    print "A"

elif subj_id == "p1001":
    print "B"
```

[Back to top](#)

Programming for Psychology in Python

Fundamentals

0. [Introduction](#)
 1. [Getting started](#)
 2. [Data—numbers, variables, and functions](#)
 3. [Data—strings and booleans](#)
 4. [Data—lists](#)
 5. [Flow control—conditionals](#)
 6. **Flow control—loops**
 7. [Dealing with errors](#)
 8. [Exercise solutions](#)
-

Flow control—iteration

Objectives

- Know how to use `for` and `while` statements for repeated execution.

Screencast

Iteration

In this lesson, we will look more at how we can control the flow of our programs. We will investigate how we can execute statements multiple times, using the `for` and `while` commands in Python.

Once again, we will consider an example situation involving an experiment in which multiple trials are presented. This experiment is extremely simple, and only consists of the trial number being printed to the screen.

If we have five trials, we could achieve this by writing:

```
print 1
print 2
print 3
```

```
print 4
print 5
```

```
1
2
3
4
5
```

Obviously, this is going to get very tedious and error-prone, particularly if there are lots of trials or the code for each trial is complex. Since it is likely that both these factors will be true for any real experiment that we would run, we need to find a way to automate these repeated tasks.

We can do so using a `for` loop. This construct takes a collection of data and loops over the items one at a time. For example:

```
trials = range(1, 6)

for trial in trials:
    print trial
```

```
1
2
3
4
5
```

As you can see, this produces the same outcome as the manual way. Let's unpack how it works. First, we used the `range` function to generate a list of the integers 1 through 5. Then, we start our loop using `for`, which is a special Python command. We then specify a variable name—as usual, this can be close to anything we like, and here we call it `trial`. We then give another special Python command, `in`, followed by our list, followed by a colon. This sequence can be read as "loop over the items in the list called `trials`, each time calling the item `trial`".

We then use indentation again to indicate nested program structure. Here, everything that is indented 'under' the `for` line is executed once for every item in the `trials` list. On the first pass through, the variable `trial` is set to the number 1, because that is the first item in the `trials` list, and the `print` statement outputs it to the screen. On the second pass through, the same process occurs except the variable `trial` is now set to the number 2, the second item in the `trials` list. This process continues until all the items in the `trials` list have been used.

The `for` loop construct is very useful, but is sometimes not ideal because we have to specify exactly how many iterations we want to perform (how many items in the list). Sometimes we don't know how many iterations we want to run—we only know that we want to continue iterating until some desired state is reached.

For example, say a trial involves presenting a dynamic pattern that the participant needs to make a judgement about. We don't know precisely when they will do this, because it is up to the participant. For a situation like this, we can use a `while` loop.

To simulate a participant in our example, we will draw a random number on each iteration of the loop. If this random number is above 0.9, we take that as akin to a participant response and we end the loop. For example:

```
import random

responded = False

while not responded:

    print "Looping!"

    if random.random() > 0.9:
        responded = True
```

```
Looping!
```

Let's work through what is happening in the above code. First, we imported the extra functionality available in the `random` package. Next, we defined a variable that is a boolean indicating whether the participant has responded yet. We then start our loop by using the special Python command `while`, followed by something that evaluates to a boolean—in this case, we are testing whether participants have not yet responded. As long as this boolean evaluates to `True`, the code that is indented beneath the `while` line will repeatedly execute. Because it starts out as `True` (because we set `responded` to `False`, and `not False` is `True`), we begin to execute the indented code. First, we print out a string just to show that we are executing this particular line of code. Next, we generate a random number between 0 and 1 and test

if it is greater than 0.9. If it is, we set the variable `responded` to the value of `True`. We then go back up to our `while` loop definition, and evaluate the boolean test again. If it continues to evaluate to `True`, we go and execute the code again. However, if it now evaluates to `False`, then we don't execute the code and we move on and finish the program.

Tip: The boolean test after the `while` command (the termination condition) needs to be carefully considered. It is very easy to produce an "infinite loop" by having a test that will never evaluate to `False`—meaning your program would run forever!

Exercises

1. Imagine that you have a 4x4 image and you want to iterate over every pixel in the image. How could you go about this using `for` loops? Hint: think about what happens when `for` loops are nested.
2. Can you loop over a string?
3. Replace the lines in the `while` loop example:

```
if random.random() > 0.9:  
    responded = True
```

with a statement that does not require an `if`.

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Fundamentals

0. [Introduction](#)
 1. [Getting started](#)
 2. [Data—numbers, variables, and functions](#)
 3. [Data—strings and booleans](#)
 4. [Data—lists](#)
 5. [Flow control—conditionals](#)
 6. [Flow control—loops](#)
 7. **Dealing with errors**
 8. [Exercise solutions](#)
-

Dealing with errors

Objectives

- Be able to interpret Python error messages.
- Know the common error types and how they arise.

Screencast

During the course of your programming, it is inevitable that you will make errors. Programming languages are generally very unforgiving of even seemingly minor departures from what they expect, and Python is no exception. When encountering such a situation, the program will tend to raise an error by displaying a message to the screen and ceasing any further processing.

Hence, an important part of programming is knowing how to interpret error messages and to fix their underlying causes. Here, we will go through some of the common errors in Python programming.

Tip: Here, we will be dealing with a class of errors that cause Python to cease functioning. There is a more insidious form of error called a *logic error*, in which the program executes without any apparent problems but the

way the program is written does not implement the desired outcome correctly.

Consequences of errors

What happens when your Python code contains an error? Here, we will have a look at what happens when we try and print the contents of a variable that we haven't defined.

```
print apples
```

```
Traceback (most recent call last):  
  File "<string>", line 1, in <module>  
NameError: name 'apples' is not defined
```

As you can see, Python prints out an error message. There are three main useful components of such messages:

1. *The line number where the error occurred.* This is not always completely accurate, but it gives you an idea of the location in your code that is causing the error.
2. *The type of error.* Python has several different classes of error; here, we have committed what is referred to as a `NameError`.
3. *The additional information.* After the type of error, the message often prints out a more detailed explanation of what went wrong. Here, it tells us that the variable name `apples` has not been defined.

Types of error—`NameError`

This class of error is encountered when Python does not recognise the name of something in a statement. As we saw above, this can happen when we try and refer to a variable that has not been defined. It was quite obvious in that example, but it can also happen because of typos:

```
apples = 2  
  
print apples
```

```
Traceback (most recent call last):  
  File "<string>", line 3, in <module>  
NameError: name 'apples' is not defined
```

You might also encounter a `NameError` if you are trying to create a string but forget to include the quotation marks (this might also give you another form of error, depending on what characters were in the string that you are trying to make):

```
hello_str = Hello
```

```
Traceback (most recent call last):  
  File "<string>", line 1, in <module>  
NameError: name 'Hello' is not defined
```

Another reason that you may encounter a `NameError` is forgetting to use an appropriate `import` statement before using additional functionality. For example:

```
print random.random()
```

```
Traceback (most recent call last):  
  File "<string>", line 1, in <module>  
NameError: name 'random' is not defined
```

Why do we get a name error here? Because we had not yet imported the `random` package:

```
import random  
  
print random.random()
```

```
0.451218119869
```

Note that these statements are executed sequentially, so the `import` statement needs to come before you try and use it:

```
print random.random()
```

```
import random
```

```
Traceback (most recent call last):  
  File "<string>", line 1, in <module>  
NameError: name 'random' is not defined
```

Types of error—SyntaxError

This class of error is encountered with Python is unable to parse a section of the code—that is, if its construction does not conform to a format that Python can interpret.

For example, Python does not know what you mean if you try and name a variable starting with a number. The way it communicates this is via a `SyntaxError`:

```
4apples = 4
```

```
File "<string>", line 1  
 4apples = 4  
   ^  
SyntaxError: invalid syntax
```

A `SyntaxError` can also be raised if you try and use an operator that Python doesn't know about in a particular context. For example:

```
print 2 $ 3
```

```
File "<string>", line 1  
print 2 $ 3  
      ^  
SyntaxError: invalid syntax
```

Finally, you may also run across a `SyntaxError` if you forget the colon in a loop or conditional:

```
if 2 > 1  
  print "A"
```

```
File "<string>", line 1  
if 2 > 1  
   ^  
SyntaxError: invalid syntax
```

Types of error—IndentationError

Python is very picky about indentation, as it needs to be—as we have seen, indentation is used to indicate program structure. As such, it needs to be used consistently throughout your code.

Our convention is to use four space characters (inserted using the `TAB` key) to indicate an indented block. You will find that if this is not used correctly, an `IndentationError` will be raised:

```
for i in range(2):  
  print i
```

```
File "<string>", line 2  
  print i  
   ^  
IndentationError: expected an indented block
```

Python here is quite useful in telling you what went wrong—it expected an indented block after a `for` statement but didn't find one. Here is a slightly different example, where Python is unable to unambiguously determine which level of indentation a statement belongs to:

```
for i in range(2):  
  for j in range(2):  
    print i  
  print j
```

```
File "<string>", line 4  
  print j
```

```
^
IndentationError: unindent does not match any outer indentation level
```

Types of error—`TypeError`

This form of error is encountered when there is something wrong with the data type that is used for a particular operation or function. We have seen this error before when we try to combine data types using an operator that it doesn't know what to do with. For example, dividing a string by a number doesn't make much sense:

```
print "Hello" / 3
```

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

We can also encounter a `TypeError` when we give a function more arguments than it is expecting. For example, the `abs` function returns the absolute value of a numerical argument. If we give it more than one, it raises a `TypeError`:

```
print abs(-3)
print abs(-3, 5)
```

```
3
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

We also get an error if we don't give a function enough arguments:

```
print abs()
```

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: abs() takes exactly one argument (0 given)
```

Types of error—`IndexError`

We encounter an `IndexError` when we try and access an element of a collection beyond the number of elements that the collection contains. For example, with a list:

```
apples = range(5)
print apples[10]
```

```
Traceback (most recent call last):
  File "<string>", line 2, in <module>
IndexError: list index out of range
```

We can also get a similar error from a string:

```
hello_str = "Hello, World!"
print hello_str[20]
```

```
Traceback (most recent call last):
  File "<string>", line 2, in <module>
IndexError: string index out of range
```

Types of error—`AttributeError`

This class of error is encountered when trying to use a function or variable attached a data (which as known as an attribute) that is not present. For example, we have seen that strings have a set of attached functions. We may have thought that there was such a function named `sum`—however, if we tried to use it we would encounter an `AttributeError`:

```
hello_str = "Hello, World!"
print hello_str.sum()
```

```
Traceback (most recent call last):
  File "<string>", line 2, in <module>
```

AttributeError: 'str' object has no attribute 'sum'

Exercises

1. Identify and correct the errors in the code below, which is intended to do five random number draws, each time printout out the draw number and "HIGH" if it is above 0.5 and "LOW" otherwise:

```
for draw in range(5):  
    print draws + 1  
    if random.random() >> 0.5:  
        draw_type = "high"  
    else  
        draw_type = "low"  
    print draw_type.UPPER()
```

2. Do you think the following would produce an `IndexError`, a `TypeError`, or no error at all? Why?

```
hello_str = "Hello, world!"  
print hello_str[2.0]
```

[Back to top](#)

[Damien J. Mannion, Ph.D.](#) — [School of Psychology](#) — [UNSW Australia](#)

Programming for Psychology in Python

Fundamentals

0. [Introduction](#)
 1. [Getting started](#)
 2. [Data—numbers, variables, and functions](#)
 3. [Data—strings and booleans](#)
 4. [Data—lists](#)
 5. [Flow control—conditionals](#)
 6. [Flow control—loops](#)
 7. [Dealing with errors](#)
 8. **Exercise solutions**
-

Exercise solutions

Scroll down to see the solutions to the exercises. The different lessons have quite large spacing so that you don't accidentally see the solutions for other lessons.

Data—numbers, variables, and functions

1. These operators implement greater than ($>$), less than ($<$), greater than or equals to ($>=$), and less than or equals to ($<=$). They each generate a boolean value (`True` or `False`).
2. Without parentheses, the multiplication operation has precedence and is evaluated first so the expression evaluates to 17. With parentheses, the addition is performed first so the expression evaluates to 25.
3. Normal division on 3.0 and 2.0 gives 1.5, but using this `//` operator gives 1.0. This suggests that it is doing some sort of rounding. Using the `//` operator on 3.5 and 2 gives 1.0 also, indicating that it does the rounding via a 'floor' operation.
4. The help for this function indicates it takes three arguments, each of which is optional—a lower limit, an upper limit, and a mode.
5. The function `math.degrees` can be used to convert radians to degrees.

Data—strings and booleans

1.

```
hello_str = "Hello, world!"  
  
print hello_str[len(hello_str) - 1]  
print hello_str[-1]
```

```
!  
!
```

2.

```
hello_str = "Hello, world!"  
  
print hello_str.replace("world", "Sydney")
```

```
Hello, Sydney!
```

3.

```
hello_str = "Hello, World!"  
  
# are all characters alphanumeric?  
print hello_str.isalnum()  
  
# are all characters alphabetic?  
print hello_str.isalpha()  
  
# are all characters digits?  
print hello_str.isdigit()  
  
# are all characters lower case?  
print hello_str.islower()  
  
# are all characters 'space's?  
print hello_str.isspace()  
  
# is the string in 'title' format?  
print hello_str.istitle()  
  
# are all characters upper case?  
print hello_str.isupper()
```

```
False  
False  
False  
False  
False  
True  
False
```

4.

```
subj_id = "p1001"  
print subj_id != "p1001"  
  
subj_id = "p1002"  
print subj_id != "p1001"
```

```
False  
True
```

5.

```
# yes!  
print 2 == 2  
print 2.0 == 2.0  
print 2 == 3
```

```
True  
True  
False
```


Data—lists

1. Many of the operators do not work with lists, raising errors when they are attempted to be used. One perhaps surprising behavior is when attempting to use an operator with a list and a number. For example, it might seem intuitive that using addition with a list and a number would add the number to each element in the list. However, this is not what happens:

```
print [1, 2] + 10
```

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

The addition operator is instead used to combine together two lists:

```
print [1, 2] + [10]
```

```
[1, 2, 10]
```

Conversely, the opposite behaviour applies with multiplication. We can multiply a list by a number, which doesn't multiply each element of the list but the entire list. We cannot, however, multiply two lists together.

```
print [1, 2] * 10
print [1, 2] * [10]
```

```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: can't multiply sequence by non-int of type 'list'
```

The take-home message is that operators applied to lists may not behave how you might expect.

2. The `remove` function deletes an element that matches a particular value from a list:

```
my_list = range(5)
print my_list

my_list.remove(2)
print my_list
```

```
[0, 1, 2, 3, 4]
[0, 1, 3, 4]
```

Something that might not be obvious is that this only deletes the *first* matching value from the list, not all of them:

```
my_list = range(5) + range(5)
print my_list

my_list.remove(2)
print my_list
```

```
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
[0, 1, 3, 4, 0, 1, 2, 3, 4]
```

Flow control—conditionals

1.

```
subj_id = "p1000"

if subj_id.startswith("p1"):
    print "A"
elif subj_id.startswith("p2"):
    print "B"
```

```
A
```

2.

```
cond_number = 5

# method 1
if cond_number >= 1:
    if cond_number <= 10:
        print "A"

# method 2
if cond_number >= 1 and cond_number <= 10:
    print "A"

# method 3
if 1 <= cond_number <= 10:
    print "A"
```

```
A
A
A
```

The first method is best avoided because the nested `if` statements are somewhat complicated. The second and third methods are both pretty readable.

3. No, it will not. Even though the expression will evaluate to `True` (because the subject ID is indeed p1001), the preceding `if` statement also evaluates to `True` (because the condition number is indeed 2). Once a statement in an `if/elif/else` block evaluates to `True`, the others are not tested.

```
cond_number = 2
subj_id = "p1001"

if cond_number == 2:
    print "A"

elif subj_id == "p1001":
    print "B"
```

```
A
```

Flow control—loops

1.

```
for x in range(4):
    for y in range(4):
        print "Using pixel at x, y: ", x, y
```

```
Using pixel at x, y: 0 0
Using pixel at x, y: 0 1
Using pixel at x, y: 0 2
Using pixel at x, y: 0 3
Using pixel at x, y: 1 0
Using pixel at x, y: 1 1
Using pixel at x, y: 1 2
Using pixel at x, y: 1 3
Using pixel at x, y: 2 0
Using pixel at x, y: 2 1
Using pixel at x, y: 2 2
Using pixel at x, y: 2 3
Using pixel at x, y: 3 0
Using pixel at x, y: 3 1
Using pixel at x, y: 3 2
Using pixel at x, y: 3 3
```

2. Yes. When looping over a string, the looped variable is assigned each character in the string in turn:

```
hello_str = "Hello, world!"

for hello_character in hello_str:
    print hello_character
```

```
H
e
l
l
o
,
 
w
o
r
l
d
!
```

3. Rather than using the `if` statement to set the boolean value in `responded`, we can simply apply a mathematical operator:

```
import random

responded = random.random() > 0.9
```

This will give `responded` a value of `True` if the random number was greater than 0.9, and `False` otherwise—the same as what would have happened using the `if` construct.

Dealing with errors

1.

```
# NameError: forgot to import random
import random

for draw in range(5):

    # NameError: it is draw rather than draws
    print draw + 1

    # SyntaxError: using both < and >
    if random.random() > 0.5:
        draw_type = "high"
    # SyntaxError: forgot the :
    else:
        draw_type = "low"

    # AttributeError: the function is called upper
    print draw_type.upper()
```

```
1
HIGH
2
LOW
3
HIGH
4
HIGH
5
LOW
```

2. This doesn't seem likely to produce an `IndexError`, because there are clearly more than 2 elements in the list. It is possible that it may not produce an error at all, since there is indeed an element at index 2. However, having it as a decimal is weird—how would it know what to return if you asked for the list index 2.5? Hence, this seems like a `TypeError`, because lists should be indexed by integers to avoid this confusion.

```
hello_str = "Hello, world!"

print hello_str[2.0]
```

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
TypeError: string indices must be integers, not float
```

[Back to top](#)