Week 1 content

Programs, languages and Python

Programs

A **computer program** (or **code**) is a set of instructions for a computer to follow. **Computer programming** (or **coding**) is the activity of writing computer programs.

Languages

When you write a program you must use a language that the computer can understand. There are many such languages - Java, Python, R, C, Ruby, JavaScript, PHP, and many, many more. Each language has its advantages and disadvantages - there is no "best" language that will be the most appropriate choice in all situations.

Python

In this course you will be learning Python. Python is one of the most popular languages, for several reasons:

- It's a general-purpose language that can be used in a wide variety of situations.
- It's mature enough that many comprehensive libraries for many disciplines have been built and extensively tested.
- It was developed recently enough to be built upon many of the fundamental programming principles that have been developed and fine-tuned over the years.
- It has a human-friendly syntax that is easy to learn.

Python has become one of the most popular languages for working with data. It has data processing and visualisation libraries such as pandas and Matplotlib that make handling data very easy (you will learn them in this course).

Even though you will be learning just one language, many of the fundamental principles you will learn, such as controlling program flow, manipulating data, and modularising code, carry over to almost every other language.

Version

Like natural languages, programming languages tend to change over time. You'll be using the latest version of Python throughout this course - Python 3. You may encounter some Python 2 code from time to time. The language differences between these two are minor, however they can be incompatible with each other.

Programming basics

Here is a very simple Python program:

▸ Run PYTHON

```python
print('Hello, world!')
```

Execution

You can **execute** the program (or **run** it) by clicking 'Run'. You should see "Hello, world!" printed just below the code. You should also see "Program exited with code 0" - this just tells you that the program executed without a problem.

When you run a program like this, the Python **interpreter** executes each line of the program, one at a time, from beginning to end (in this case there is only one line).

Whenever you see a runnable piece of code like this you can also modify it yourself. This is a great way for you to experiment with Python (don't worry, you won't break anything, and everything gets restored when you refresh the page). Try modifying the code above to get it to print your name rather than "world".

Statements

Each line the program is called a **statement**. Your programs will typically contain many statements. Here is a program with two statements (click the "Run" button):

▸ Run PYTHON

```python
print('Hello, world!')
print('Goodbye, world!')
```

You normally write each statement on its own line, as above. But you can write two or more statements on the one line, as long as you separate them by semicolons:

▸ Run PYTHON

```python
print('Hello, world!'); print('Goodbye, world!')
```

However, this can make a program more difficult to read, and it's recommended that you don't do this.

Syntax

Like natural languages, such as English and French, programming languages have a specific grammar; this is known as the language's **syntax**. Unlike natural languages, programming languages require you to be precise and unambiguous. A big part of learning a language is learning its syntax.

Whitespace

Python will ignore blank lines between statements and spaces between arguments in a statement. So the following code snippets are equivalent:

▸ Run PYTHON

```

print   ( 'Hello'    )

print(   10     *       2)
```

▸ Run PYTHON

```
print('Hello')
print(10*2)
```

Spacing at the start of a line is <u>not</u> ignored by Python. It is called **indentation**, and is a very important aspect of Python syntax. You will learn more about indentation when we cover `if` and `while` statements. For now you just need to know that if you incorrectly add indentation, Python will generate an error.

▸ Run PYTHON

```
    print('Hello')
```

Spacing inside quotes is also <u>not</u> ignored by Python:

▸ Run PYTHON

```
print('Hello      world')
```

Although much of your whitespace is ignored by Python it has become conventional to use it in a certain way, described in a set of guidelines called "The PEP 8 Style Guide", or just "PEP 8". According to PEP 8, for example, you should use `print('Hello')` rather than `print( 'Hello' )` or `print ( 'Hello' )`.

There is a link to PEP 8 in the Orientation section of this course, for your information. The guidelines are comprehensive, covering much more than just the use of whitespace. You're not expected to know all of them. If you just follow the formatting used in the examples throughout this course then your code should be fine.

Case sensitivity

Python is a **case-sensitive** language. This means that "print" and "Print" are considered to be different words. See what happens when you try to use "Print" instead of "print":

Run

PYTHON

```
Print('Hello world!')
```

Output and input

Output

As you've seen, you can use `print` in a program to produce user output:

Run PYTHON

```python
print('Hello, again!')
```

A very handy technique when using `print` is to use an **f-string**. If you append an 'f' to the start of a quoted string (before the first quote) you can get Python to do things inside the string before it prints, such as perform calculations. You do this by using curly brackets. Here are some examples:

Run PYTHON

```python
print(f'The result is {10*2}.')
print(f'The results are {10*2}, {10*3}, and {10*4}.')
```

Input

You can also get user input, using `input`:

Run PYTHON

```python
input('What is your name? ')
```

Python pauses until the user hits enter, and then continues executing the program. The example above is not very interesting because the program does nothing with the input it gets. Here's a more interesting example, in which the input gets printed:

Run PYTHON

```python
print(input('What is your name? '))
```

It's more interesting if we do something sensible with the input. The following uses an f-string to greet the user:

Run PYTHON

```python
print(f"Hello, {input('What is your name? ')}!")
```

Calling functions

You have seen how to use `print` and `input`. These are both **functions**. Functions are a very important part of programming. You will learn many different Python functions, and you'll even learn how to create your own.

Calling a function

You **call** a function, or **invoke** it, but putting round brackets after it. Notice the round brackets in the following:

Run PYTHON

```python
print('Hello world!')
input('What is your name? ')
```

Arguments

Inside the round brackets you can provide the function with **arguments**. In the examples above you have provided the `print` function with one argument - the string `'Hello world!'`. And you have provided the `input` function with one argument also - the string `'What is your name? '`. Each function does something with your argument.

Some functions don't expect any arguments, and will just ignore any you might provide. Some functions demand one or more arguments, and will raise an error if you don't provide them - these are called **required arguments**. Some functions don't expect any arguments but allow you to optionally provide them if you like, and will use them if you do - these are called **optional arguments**. Some functions have a mixture of required arguments and optional arguments. Part of learning a function is learning what arguments it requires and allows.

What about `print` and `input` - do they require any arguments? You can try and see:

Run PYTHON

```python
print()
input()
```

No, no arguments are required (we didn't get an error).

Multiple arguments

To provide a function with multiple arguments you just separate them by commas. The `print` function allows you to optionally provide multiple arguments. Let's see what happens if you do:

▸ Run PYTHON

```python
print('The', 'cat', 'sat', 'on', 'the', 'mat')
```

It prints each of the arguments, separated by spaces.

Many functions expect you to provide arguments in a very particular order. Again, part of learning the function is learning this order.

Providing arguments by keyword

Remembering the correct order of arguments can be hard. Fortunately, many functions allow you to provide arguments **by keyword**, which means that you can tell the function which argument is which by naming them, rather than putting them in a particular order.

The `print` function, for example, has some optional arguments that you can provide by keyword. You can use the keyword `sep` to specify what string to use to separate the items that get printed:

▸ Run PYTHON

```python
print('A', 'B', 'C', sep=', ')
```

And you can use the `end` keyword to specify a string to add at the end:

▸ Run PYTHON

```python
print('The', 'cat', 'sat', 'on', 'the', 'mat', end='.')
```

You can use them together, and because you are using keywords you can put the two in any order:

▸ Run PYTHON

```python
print('The', 'cat', 'sat', 'on', 'the', 'mat', end='.', sep='-')
print('The', 'cat', 'sat', 'on', 'the', 'mat', sep='-', end='.')
```

Return values

When you call a function, it always **returns** a value. At the very least it returns the special value `None` (you'll learn what this is), but typically it returns something more interesting.

The `input` function returns the string that the user inputs. You can do whatever you want with this string, including ignoring it. In the program below, the first line ignores it, the second line prints it:

▸ Run PYTHON

```python
input('What is your name? ') # Return value ignored
print(input('What is your name? ')) # Return value printed
```

Notice what's happening in that second line: The string `'What is your name? '` is provided as an argument to a call of the `input` function, and the string that is returned is then provided as an argument to a call of the `print` function.

> One thing to keep in mind when using the `input` function is that it always returns a string, even if the user inputs numbers. You will learn more about this, and the importance of it, in later slides.

What does the `print` function return? It's easy to find out, just print it:

Run PYTHON

```
print(print('Hello, world!'))
```

As you can see, it returns `None`.

Notice what's happening in this example: The string `'Hello, world!'` is provided as an argument to a call of the `print` function; that function prints the string and returns `None`; `None` is then provided as argument to another call of the `print` function, which prints it. That's why two things get printed - first `Hello, world!`, then `None`.

Encountering errors

We need to talk about errors early on, because errors are a fact of programming life.

There are three main types of error that you might encounter when writing and running programs:

- **Syntax errors**
- **Runtime errors**
- **Logical errors**

Syntax errors

Before executing a program the Python interpreter first checks whether the program has correct syntax. If it finds any syntax errors it will tell you, and not run the program. The following program contains a syntax error (it's missing a quotation mark in line 2) - try running it:

▸ Run PYTHON

```
1 print('The program is running')
2 print('Hello, world!)
```

Notice that the first line is not executed - that's because the program never gets to run, because of the syntax error.

Runtime errors

Sometimes the syntax of a program is fine, but Python raises an error during its execution. These are called "runtime errors". When this happens, execution stops, and Python returns some information about the error.

For example, the following program asks Python to divide a number by zero in line 2, which is impossible. It generates a runtime error:

▸ Run PYTHON

```
1 print('The program is running')
2 print(10/0)
```

Notice that the first line is executed this time - that's because the programs runs, at least until it gets to line 2.

Logical errors

Sometimes the syntax of a program is fine, and it executes without any runtime errors, but it doesn't do what you intended it to do. This is called a "logical error".

For example, suppose you write the following program to add 1 and 2 and then multiply the result by 3:

Run PYTHON

```python
print(1 + 2 * 3)
```

The result you're expecting is 9, because 1 + 2 is 3, and 3 times 3 is 9. But the program above prints 7. Error! This is not a syntax error, nor a runtime error - it's a logical error. The problem is that Python assumes that you want the multiplication to happen first, then the addition (you will learn more about this). So it calculates 2 x 3 = 6, then 1 + 6 = 7. You should write you program like this instead:

Run PYTHON

```python
print((1 + 2) * 3)
```

This removes the logical error.

Adding comments

In Python, anything after a hash symbol `#` on a line will be ignored by the Python interpreter. This allows you to do some useful things.

Explaining your code

It is important that the programs you write are easily understandable by someone who reads them. To help with this, it can be a good idea to add comments throughout.

Here is an earlier example, with an explanatory comment added:

▸ Run PYTHON

```python
# Ask the user for input and then print it:
print(input('What would you like to print? '))
```

Comments can start anywhere on a line, but comments that come after code can be difficult to read - use them sparingly:

▸ Run PYTHON

```python
print(input('What would you like to print? ')) # Ask the user for input
```

You should not use comments to state the obvious. Comments should explain the intention of your code, instead of describing the exact procedure it is performing. What the code is *doing* is often obvious, but it may not be obvious *why* it is doing it.

Here in Ed we'll add quite a few comments to code that wouldn't ordinarily be added, for instructional purposes.

Disabling your code

You can also use `#` to disable one or more lines of code, either because they are not yet finished and will cause an error, or because your program is not working and you are trying to find the cause of the error, or because you are not using them but want to save them just in case you decide to use them again. For example:

▸ Run PYTHON

```python
print('Hello, world!')
# print('Goodbye, world!')
```

Keep in mind that disabled code can be distracting, so get rid of it as soon as you can, either by enabling it again, or deleting it altogether.

Multiline comments?

Alas, the only way you can have a multiline comment in Python is to put a `#` at the start of each line. (Some languages have special techniques.)

As a workaround you could put triple quotes before the first line of code and after the last line, but this does not make the lines into a multiline comment that Python ignores - it makes them into a multiline string which Python evaluates and then does nothing with. It's not good practice.

Importing modules

You won't get far using Python before you need to **import a module**. A module is just a file that contains Python code, with a ".py" extension. Although the core Python language contains a lot of functionality, some of the functionality you'll want is included in modules, rather than in the core. You can access that functionality by importing those modules.

For example, there is a lot of non-core mathematics functionality in the "math.py" module. You can import that module as follows (note that you don't include ".py" - Python will figure it out):

Run PYTHON

```python
import math
```

This makes all of the code that has been saved in the math.py file available for use in your program. For example, in math.py there is a function `floor`, which rounds a number down to the nearest integer. Having imported math.py you can now call this function:

Run PYTHON

```python
import math
print(math.floor(3.14))
```

Note that you call the function using `math.floor`, rather than using just `floor`. If you'd like to be able to use this function without having to add the prefix, you can do so using the `from` keyword when you import:

Run PYTHON

```python
from math import floor
print(floor(3.14))
```

If you'd like to import more functions then you can just separate their names by commas:

Run PYTHON

```python
from math import floor, ceil
print(floor(3.14))
print(ceil(3.14))
```

You can import everything by using an asterisk:

Run PYTHON

```python
from math import *
print(floor(3.14))
print(ceil(3.14))
```

But you should avoid doing this because you might end up with the same function name being used twice.

You can use the `dir` function to see what is available to you in a module that you have imported:

▸ Run PYTHON

```python
import math
print(dir(math))
```

When you import a module you can give it an alias using the `as` keyword, to save yourself some typing. One module that we will be using a lot in the second half of this course is the pandas module. It is standard to give it the alias "pd" when importing it:

▸ Run PYTHON

```python
import pandas as pd
print(dir(pd))
```

When Python gets installed on a computer many modules are installed with it. These are called **built-in modules**. These are automatically available for you to import. If you want to import other modules they will first need to be installed on the computer. You can also write your own modules, and import them - you will learn how to do that in Week 3.

Objects

When you program with Python (and many other languages) you work a lot with **objects**. In Python, just about everything is an object. The number 10 is an object, the string "Hello, world!" is an object, even the `print` function is an object.

Types

Every object is of a certain **type**. In Python, the main types are:

- **Integer (int)**. A whole number, positive or negative, including zero (i.e. ..., -2, -1, 0, 1, 2, ...).
- **Floating-point number (float)**. A positive or negative number, not necessarily whole, including zero (e.g. 3.14, -0.12, 89.56473).
- **String (str)**. A sequence of characters (e.g. 'Hello', 'we34t&2*').
- **Boolean (bool)**. A truth value, either true or false.
- **List (list)**. An ordered collection of objects.
- **Tuple (tuple)**. An immutable list (i.e. one that cannot be changed).
- **Set (set)**. An unordered collection of unique objects.
- **Dictionary (dict)**. A set of key-value pairs.
- **Function**. A piece of code that can be run by calling it.
- **Class**. A user-defined type of object.

Python also has a special object, `None`, which represents the absence of an object. It is of type **NoneType**. It is the only object of this type.

You will be learning more about the objects of each type. This week you will learn about integers, floats, strings, and booleans. Next week you will learn about lists, tuples, sets, and dictionaries. In Week 3 you will learn about functions and classes.

Checking the type of an object

You can check the type of an object by using Python's `type` function:

▸ Run PYTHON

```python
print(type(1))
print(type(3.14))
print(type('Hello'))
print(type(None))
print(type(print))
```

Notice in the last example above that we have provided the `print` function as an argument to the `type` function. You can do that - the `print` function is an object, just like numbers and strings are, and you can provide it as an argument to the `type` function to find out what type of object it is. Most of the time you'll be calling `print`, and providing arguments to it, but occasionally you might provide it as an argument to some other function, as we have done above. In fact, you can even provide `print` as an argument to itself:

Run PYTHON

```python
print(print)
```

When an object is of a certain type we say that it is an **instance** of that type. You can also use Python's `isinstance` function to check whether an object is of a certain type:

Run PYTHON

```python
print(isinstance(1, int)) # True
print(isinstance(1, float)) # False
print(isinstance(3.14, float)) # True
print(isinstance(3.14, int)) # False
print(isinstance('Hello', str)) # True
print(isinstance('Hello', float)) # False
```

Attributes

Objects have **attributes**.

Each attribute is either a function or a non-function. If it's a function, it's called a **method** of the object. You can think of this as something that you can do with the object. If it's a non-function, it's called a **field** of the object. You can think of this as some fact about the object. So attributes come in two varieties:

- Methods - things you can do with the object
- Fields - facts about the object

Each attribute has a name, and you can refer to an object's attribute using **dot notation.** For example, a string object has an attribute `upper`. You can refer to this attribute of the string "Hello" like this: `"Hello".upper`. It also has an attribute `lower`, to which you can refer like this: `"Hello".lower`.

Both of these attributes are functions, so both are methods. Since they are functions you can invoke them using round brackets `()`, like this: `"Hello".upper()` , `"Hello".lower()`. The first is telling Python to invoke the `upper` method of the `"Hello"` object; the second is telling Python to invoke the `lower` method.

What do these two methods do? The first returns the string in upper case, the second returns the string in lower case:

▸ Run PYTHON

```python
print("Hello".upper())
print("Hello".lower())
```

Note that every string object has these two methods. But when you invoke these methods they might return different values for different strings - it depends on which string object they are invoked on:

▸ Run PYTHON

```python
print("Hello".upper())
print("Goodbye".upper())
```

Which attributes an object has depends upon what type of object it is. String objects have the `upper` attribute, but function objects do not. If you try to access this attribute of a function you will get an error:

▸ Run PYTHON

```python
print("Hello".upper())
print(input.upper())
```

Part of learning Python, and other languages, is learning what attributes the different types of objects have.

Expressions

To work with an object you need to refer to it, and to refer to it you use an **expression**.

Expressions come in two varieties: **simple expressions** and **complex expressions**. We'll consider simple expressions first, and then complex expressions.

There are three types of simple expressions:

- **Literals**
- **Variables**
- **Constants**

Literals

A **literal** shows explicitly which object it refers to. Here are some examples:

- Integer literals: `1`, `26`, `0`
- Floating-point literals: `3.14`, `0.06`, `0.0`, `2.4e12`, `5.12E-2`
- String literals: `'Hello'`, `"Principles of Programming"` (you can use single or double quotes, but they must match)
- Boolean literals: `True`, `False` (there are only two)
- None literal: `None` (there is only one)

Variables

You can also introduce your own names for objects, and these are called **variables**.

When you introduce a variable you have to specify which object it refers to. This is called **assigning** the variable, and you do it using the **assignment operator** `=`. This is sometimes also called **binding** the variable - you bind it to an object.

Here's an example:

Run PYTHON

```python
message = 'Hello there' # Introduce a variable and assign it an object
print(message)
```

The first line introduces a variable `message` and assigns it the string object `'Hello there'`. The second line uses this variable to print the object.

Once you introduce a variable you can use it as many times as you like throughout your program:

▸ Run PYTHON

```python
message = 'Hello there'
print(message)
print(message)
print(message)
print(message)
```

The object that the variable refers to is often called the **value** of the variable. You can change the value of a variable (i.e. change which object the variable refers to) as often as you like:

▸ Run PYTHON

```python
message = 'Hello there'
print(message)
message = 3.14 # Assign a new value
print(message)
message = True # Assign a new value
print(message)
```

Notice that the variable `message` in this program refers to different types of object as the program proceeds - first it refers to a string, then to a floating-point number, and then to a boolean. Because variables in Python can do this we say that Python has **flexible typing**. (Some languages don't allow variables to change their reference to a different type of object.)

If you want to check the type of the object that a variable refers to, you can use the `type` function:

▸ Run PYTHON

```python
x = 'Hello there'
print(type(x)) # str
x = 3.14
print(type(x)) # float
x = True
print(type(x)) # bool
x = print
print(type(x)) # function
```

Any type of object can be assigned to a variable, not just numbers, strings or booleans. Look at the last example above - we have assigned the `print` function to the variable `x`. Having done that, `x` is then another name for the `print` function, and we can use it accordingly:

▸ Run PYTHON

```python
x = print
x('Hello, world!')
```

You can use variables to assign other variables:

Run PYTHON

```python
1 message = 'Hello there'
2 new_message = message
3 print(new_message)
```

Be careful to understand what's going on here. In line 2 you are getting `new_message` to refer to whatever it is that `message` refers to at the time. If a different object is later assigned to `message`, it will <u>not</u> automatically be assigned to `new_message` as well - there will be no change in what `new_message` refers to. Let's check:

Run PYTHON

```python
message = 'Hello there'
new_message = message
message = 'Goodbye' # Value of new_message not changed
print(new_message)
```

You must assign a value to a variable before you use it, otherwise Python will generate an error:

Run PYTHON

```python
print(message) # Error - message has no value
```

If you want to introduce a variable but don't yet have any significant value to assign it then you can assign it the value `None`:

Run PYTHON

```python
message = None
print(message) # No error - message has a value
```

You can assign multiple variables the same value:

Run PYTHON

```python
x = y = z = 0 # All get assigned 0
print(x, y, z)
```

Or different values, using a technique called **unpacking**:

Run PYTHON

```python
x, y, z = 1, 2, 3 # x gets 1, y gets 2, z gets 3
print(x, y, z)
```

This technique gives you a way to swap the values of two variables:

Run PYTHON

```python
x, y = 1, 2
print(x, y)
x, y = y, x # Swap the values of x and y
print(x, y)
```

Naming rules

You have a fair bit of freedom in what names you can introduce, but there are some rules and restrictions.

First, you cannot use any of Python's keywords, such as `import`, `from`, `True`, `False`, `None`, etc. There are 33 keywords in version 3 - you will be learning most of them.

Second, a name can only contain uppercase letters (`A-Z`), lower case letters (`a-z`), digits (`0-9`) and the underscore (`_`). Moreover, the first character cannot be a digit.

Keep in mind that Python is case sensitive, and this applies to your names as well. The program below generates an error because the variable `Message` is not given a value before it is used (the variable `message` is, but that's a different variable):

Run PYTHON

```python
message = 'Hello there'
print(Message) # Error - Message does not have a value
```

It is best to choose names for your variables that make the intention of your program as clear as possible. Consider the following two pieces of code:

```python
var1 = 10
var2 = 120
print(var1 * var2)
```

```python
days = 10
fish_per_day = 120
print(days * fish_per_day)
```

Both pieces of code do the same thing, but the second makes it much clearer what is going on. In effect, by choosing variable names carefully we can use them to help explain the code.

It is fairly standard to make variable names lowercase, with words separated by underscores, e.g, `fish_per_day`. This is often called **snake case**.

Unbinding variables

You can unbind a variable from an object by using `del`.

▸ Run PYTHON

```python
x = 10
print(x)
del x # Unbind x from its value
print(x) # Now we get an error
```

Note that the word "del" is a bit misleading. You are not deleting the variable or the object it is bound to - you are just severing the connection between them.

Although, when you do this the variable and/or the object might get removed from memory, via a process called **garbage collection**. An object is removed from memory by Python when nothing is referring to it.

Also note that unbinding a variable is not the same as **rebinding** it to `None`:

▸ Run PYTHON

```python
x = 10
print(x)
x = None
print(x) # No error - x has a value
```

Choosing types

You can get Python to return an object of a certain type by using the functions `int`, `float`, `str`, `bool`, `list`, `tuple`, `set`, and `dict`. For example, if you would like `var` to refer to the floating-point number 1 rather than the integer 1 you can use the `float` function:

▸ Run PYTHON

```python
var = 1 # var refers to the integer 1
print(var, type(var))

var = float(1) # var refers to the floating point number 1.0
print(var, type(var))
```

You might need to do this when you are getting user input. Python treats user input as a string, so if you are asking the user to enter a number then you will need to convert the input, using `int` or `float`:

Run PYTHON

```python
# number will be a string:
number = input('Enter a number: ')
print(number, type(number))

# number will be an integer:
number = int(input('Enter a number: '))
print(number, type(number))

# number will be a float:
number = float(input('Enter a number: '))
print(number, type(number))
```

Sometimes Python can't return an object of the type you are asking for, and it will raise an error. For example, Python cannot make every string into an integer:

Run PYTHON

```python
var = int('hello') # Error - cannot make 'hello' into an integer
print(type(var))
```

You can use the same functions to change the type of object a variable refers to after it has been set:

Run PYTHON

```python
var = 2 # var refer to the integer 2, by default
print(var, type(var))

var = float(var) # var now refers to the floating-point number 2.0
print(var, type(var))

var = str(var) # var now refers to the string '2.0'
print(var, type(var))
```

Note that when it operates on a floating-point number the `int` function truncates all decimal places:

Run PYTHON

```python
print(int(1.2))
print(int(-1.2))
```

Constants

If you intend the value of a variable not to change, then you are using it as a **constant**. It is conventional to indicate this by naming it using all capital letters, with underscores separating the words, e.g, `MAX_INT`. One of the main reasons for using constants is to give an indication as to *why* a particular value is being used (e.g. using the constant HOURS_PER_DAY instead of the literal 24), so again, it's best to use names that help to explain what your program does:

Run PYTHON

```python
HOURS_PER_DAY = 24 # Signal that this should not change
MINUTES_PER_HOUR = 60 # Signal that this should not change

num_days = int(input('How many days? '))
num_minutes = num_days * HOURS_PER_DAY * MINUTES_PER_HOUR

print(num_days, 'days is', num_minutes, 'minutes')
```

Note that just with any other variable, it is *possible* to change the value of a constant - it is up to the programmer to ensure that variables that are intended to be constants do not change after they are defined.

Complex expressions

A **complex expression** is an expression that contains subexpressions, combined using arithmetical and other kinds of operators (you will be learning about various kinds of operators).

For example, `10 * 2`. This is an expression, because it refers to an object - the number 20. And it is a complex expression, because it contains a subexpression - the literal `10`. Actually, it contains another subexpression too - the literal `2`. But it only needs to contain one subexpression to count as complex. These two subexpressions are combined using the `*` operator.

You will see many examples of complex expressions as we proceed.

Working with numbers

Python distinguishes two types of numbers:

- Integers
- Floating point numbers

There are also complex numbers, but we won't be considering them in this course.

Operating on numbers

You can add, subtract, multiply, and divide numbers by using the **operators** `+`, `-`, `*`, `/`, respectively, and combinations of them:

Run PYTHON

```
print(1 + 2)
print(10 - 5)
print(3 * 4)
print(20/4)
print((1 + 2)*(3 + 4))
print(10/(3-1))
```

You can raise one number to the power of another number by using the **exponentiation operator**, `**`:

Run PYTHON

```
print(10 ** 2) # 10 to the power 2
print(2 ** 3)  # 2 to the power 3
```

You can divide one number by another number and round down to the nearest whole number by using the **integer division operator**, `//`:

Run PYTHON

```
print(10 // 3) # 10 divided by 3 and rounded down
```

Note the effect of using `//` on negative numbers:

Run PYTHON

```
print(-10 // 3) # -10 divided by 3 and rounded down
```

You can divide one number by another number and get the remainder by using the **modulus operator**, `%`:

▸ Run PYTHON

```
print(10 % 3) # The remainder when 10 is divided by 3
```

Order of operations

Unless you specify the order in which operations are to be performed, by using brackets, Python performs them in a very particular order:

- First, `**` is performed, from right to left
- Next, `*`, `/`, `//`, and `%` are performed, from left to right.
- Next, `+` and `-` are performed, from left to right.

▸ Run PYTHON

```
print(1 + 2 * 3) # Same as 1 + (2*3)
print(2 ** 3 * 4) # Same as (2**3) * 4
print(24 / 6 * 2) # Same as (24/6) * 2
print(24 / 6 / 2) # Same as (24/6) / 2
print(2 ** 2 ** 3) # Same as 2**(2**3)
```

Augmented assignment operators

You will often find yourself wanting to operate on a variable and then re-assign the result to that same variable. Python provides **augmented assignment operators** to allow for more concise code: `+=`, `-=,` ,`*=`, `/=`, `//=`, and `%=` .

- `x += 2` is equivalent to `x = x + 2`
- `x -= 2` is equivalent to `x = x - 2`
- `x *= 2` is equivalent to `x = x * 2`
- `x /= 2` is equivalent to `x = x / 2`
- `x //= 2` is equivalent to `x = x // 2`
- `x %= 2` is equivalent to `x = x % 2`

▸ Run PYTHON

```
x = 12
x += 2 # Equivalent to x = x + 2 (assign to x the result of x + 2)
print(x)
```

Comparing numbers

Python has a number of **comparison operators** which you can use to compare numbers:

- `num1 == num2` is `True` if the value of `num1` **equals** the value of `num2`, otherwise it is `False`
- `num1 is num2` is `True` if the value of `num1` **equals** the value of `num2`, and `num1` and `num2`

have the same type, otherwise it is `False`

- `num1 != num2` is `True` if the value of `num1` **does not equal** the value of `num2`, otherwise it is `False`
- `num1 is not num2` is `True` if the value of `num1` **does not equal** the value of `num2`, or `num1` and `num2` **have different types**, otherwise it is `False`
- `num1 < num2` is `True` if the value of `num1` **is less than** the value of `num2`, otherwise it is `False`
- `num1 <= num2` is `True` if the value of `num1` **is less than or equal to** the value of `num2`, otherwise it is `False`
- `num1 > num2` is `True` if the value of `num1` **is greater than** the value of `num2`, otherwise it is `False`
- `num1 >= num2` is `True` if the value of `num1` **is greater than or equal to** the value of `num2`, otherwise it is `False`

▸ Run PYTHON

```python
print(1 == 1.0)
print(1 is 1.0)
```

Floating-point numbers and `==`

Because floating point numbers are stored with limited precision you might experience strange results when attempting to compare them with `==`. For example:

▸ Run PYTHON

```python
print(0.1 + 0.2 == 0.3)
```

Why does this comparison return `False`? Because `0.1 + 0.2` is not what you would expect:

▸ Run PYTHON

```python
print(0.1 + 0.2)
```

When you compare floating point numbers using `==` or `!=` you should always round them float to a specified precision, using the `round` function: `round(x, n)` rounds the value of `x` to n decimal places.

▸ Run PYTHON

```python
print(round(0.1 + 0.2, 1))
print(round(0.3, 1))
print(round(0.1 + 0.2, 1) == round(0.3, 1))
```

Note: the `round` function implements round half to even:

Run PYTHON

```python
print(round(4.5,0))
print(round(5.5,0))
print(round(-4.5,0))
print(round(-5.5,0))
```

Applying functions

Python has the functions `int`, `float`, `abs`, `pow`, `round` that you can use to manipulate numbers:

Run PYTHON

```python
print(int('3') + float('2.0')) # Convert strings to numbers and add them
print(abs(-5)) # Absolute value of -5
print(pow(2, 4)) # 2 to the power of 4.  This is the same as 2**4
print(round(3.567, 2)) # Round 3.567 to 2 decimal places
```

Generating random numbers

It can be very useful to generate random numbers. You can do this by importing the functions `random` and `randint` from the `random` module.

Run PYTHON

```python
from random import random, randint
print(random()) # Random float between 0 and 1
print(randint(10, 20)) # Random integer between 10 and 20 (inclusive)
```

Other mathematical functions

A lot of other mathematical functions that you might need are in in `math` module:

Run PYTHON

```python
import math
print(math.sin(0.5)) # Degrees are in radians
print(math.cos(0.5))
print(math.tan(0.5))
print(math.sqrt(225)) # Square root
print(math.log(10)) # Natural logarithm, base e
print(math.ceil(2.05)) # Round up to nearest integer
print(math.floor(2.95)) # Round down to nearest integer
```

Working with strings

A string is a sequence of characters. The characters might be ones you can type on your keyboard, or any of the hundreds of thousands of other **unicode characters**, including symbols and foreign language letters.

String literals

As you have seen, string literals have the characters between quotes, either single quotes or double quotes.

▸ Run PYTHON

```
literal_one = 'A string' # You can use single quotes
literal_two = "A string" # Or double quotes
print(literal_one)
print(literal_two)
```

The empty string

One of the most useful strings is the **empty string** - a string with no characters. The literal for an empty string is `''`, or `""`.

Note that the empty string is not the same thing as `None` - they are different objects:

▸ Run PYTHON

```
print('') # Empty string
print(None) # None
print('' is None) # They are not the same
```

Escaping special characters

Sometimes you will need to use a string literal that contains quote marks or other characters that have special meaning in Python. You can do this by **escaping** those special characters, which means prefixing them with a backslash `\`.

Run PYTHON

```python
# Backslash used to escape a quote mark
print('Penny\'s dog')
print("Penny's dog is called \"Mac\".")

# Backslash used to escape a new line symbol
print('This is one line.\nThis is a second line')

# Backslash used to escape a tab symbol
print('Here\tThere')
```

This means that the backslash itself has a special meaning in strings, so to include it you must escape it:

Run PYTHON

```python
print("You can have tea neither\nor coffee")
print("You can have tea neither\\nor coffee")
```

If you have a lot of backslashes and no special characters then you can tell Python to ignore the special meaning of the backslash by preceding the string with `r` (for "raw"):

Run PYTHON

```python
print(r"You can have tea neither\nor coffee")
```

You can avoid having to escape single quote marks by enclosing the whole string in double quotes. Similarly, you can avoid having to escape double quote marks by enclosing the whole string in single quotes:

Run PYTHON

```python
print("Penny's dog")
print('Penny said, "This is my dog".')
```

If you like to include line breaks in a string then you can either use `\n`, as in the example above, or you can use triple quotes around the string:

Run PYTHON

```
# Using \n
text = 'This is one line.\nThis is a second line'
print(text)

# Using triple single quotes
text = '''This is one line.
This is a second line.'''
print(text)

# Using triple double quotes
text = """This is one line.
This is a second line."""
print(text)

```

You can also use triple quotes around a string that contains both single and double quotes:

Run PYTHON

```
text = '''She said, "I don't know how you do it!"'''
print(text)
text = """She said, 'I don't know what "" means'"""
print(text)
```

You'll get an error if you end up with four quotes in a row.

Concatenating strings

You can **concatenate** strings (i.e. join them) using the `+` operator:

Run PYTHON

```
first_name = 'Leo'
last_name = 'Tolstoy'
full_name = first_name + ' ' + last_name
print(full_name)
```

You can also use the `+=` augmented assignment operator:

Run PYTHON

```
name = 'Leo'
name += ' '
name += 'Tolstoy'
print(name)
```

You can duplicate a string a given number of times by using the `*` operator:

Run PYTHON

```python
print('a' * 10)
```

Comparing strings

Just as with numbers, you can use Python's comparison operators to compare strings:

- `str1 == str2` is `True` if `str1` and `str2` are the same sequence of characters, otherwise it is `False`
- `str1 != str2` is `True` if `str1` and `str2` are not the same sequence of characters, otherwise it is `False`
- `str1 in str2` is `True` if `str1` appears as a substring in `str2`, otherwise it is `False`
- `str1 not in str2` is `True` if `str1` does not appear as a substring in `str2`, otherwise it is `False`
- `str1 < str2` is `True` if `str1` **is lexicographically less than** (i.e. would appear earlier in the dictionary than) `str2`, otherwise it is `False`
- Similarly we have `str1 <= str2`, `str1 > str2`, and `str1 >= str2`

Run PYTHON

```python
print('A' == 'A')
print('A' == "A") # Whether you define literals with ' or " does not mat
```

Run PYTHON

```python
print('fish' in 'selfishness')
print('fine' in 'selfishness') # Substrings have to be contiguous
```

Run PYTHON

```python
print('A' < 'B')
print('AA' < 'AB')
print('A' < 'AA')
```

Applying functions to strings

You can use the `len` function to find the length of a string:

Run PYTHON

```python
print(len('abcde'))
```

Calling string methods

String objects have many useful methods. For example, they have a method `upper` which returns a string with the same characters but all in upper case.

▸ Run PYTHON

```python
s = 'hello'
print(s.upper())
```

None of a string's methods **modify the string in place** - they all **return a new string**. So, if you want to modify a string by using one of the methods you have to assign the result of the method back to the string. To illustrate:

▸ Run PYTHON

```python
s = 'hello'
print(s)

s.upper() # Returns a new string - does not change s
print(s)

s = s.upper() # Assign the new string back to s - changes s
print(s)
```

Some of the most useful string methods are listed below.

- `str.isupper` - Returns true if all characters in `str` are upper case
- `str.islower` - Returns true if all characters in `str` are lower case
- `str.isalpha` - Returns true if all characters in `str` are from the alphabet
- `str.isdigit` - Returns true if all characters in `str` are digits
- `str.isnumeric` - Returns true if all characters in `str` are numeric
- `str.isspace` - Returns true if all characters in `str` are whitespace characters (i.e. space, tab, or new line)
- `str.startswith` - Returns true if `str` starts with the specified value
- `str.endswith` - Returns true if `str` ends with the specified value
- `str.upper` - Returns `str` with every character in upper case
- `str.lower` - Returns `str` with every character in lower case
- `str.title` - Returns `str` with the first character of each word in upper case
- `str.capitalize` - Returns `str` with the first character uppercase and the rest lowercase
- `str.format` - Returns `str` formatted as specified
- `str.find` - Searches `str` for a specified value and returns the index at which it is first found (or -1 if it was not found)
- `str.index` - like `str.find` but raises an error if the value is not found
- `str.rfind` - Searches `str` for a specified value and returns the index at which it is last found (or -1 if it was not found)
- `str.count` - Returns the number of times a specified value occurs in `str`
- `str.strip` - Returns `str` with whitespace stripped from both ends

- `str.lstrip` - Returns `str` with whitespace stripped from the left end
- `str.rstrip` - Returns `str` with whitespace stripped from the right end
- `str.replace` - Returns `str` with a specified value replaced by a specified value

Working with booleans

Boolean objects are the simplest type of objects in Python, but often they play the most significant role in determining the path that a program takes (as you will see).

There are two boolean objects: `True` and `False`. They most commonly appear as the result of the comparison operations we have seen for the other types, for example `3 < 5` will evaluate to `True`; and `1 + 1 == 3` will evaluate to `False`:

▸ Run PYTHON

```python
print(3 < 5) # True
print(1 + 1 == 3) # False
```

Operating on booleans

You can build complex boolean expressions by combining simple boolean expressions with the logical operators `not`, `and`, and `or`:

- `not x` is true if `x` is false, otherwise it is false.
- `x and y` is true if `x` is true and `y` is true, otherwise it is false.
- `x or y` is true if `x` is true or `y` is true or both, otherwise it is false

▸ Run PYTHON

```python
print(not True) # False
print(not False) # True
print(not 3 < 5) # False
```

▸ Run PYTHON

```python
print(True and True) # True
print(True and False) # False
print(False and True) # False
print(False and False) # False
print((3 < 5) and (1 + 1 == 3)) # False
```

▸ Run PYTHON

```python
print(True or True) # True
print(True or False) # True
print(False or True) # True
print(False or False) # False
print((3 < 5) or (1 + 1 == 3)) # True
```

Run PYTHON

```python
guess = int(input('Enter a number: '))
print('Your number is between 2 and 5?')
print(guess > 2 and guess < 5)
```

Chaining comparisons

It is possible to combine comparisons into a chain:

Run PYTHON

```python
x = 12
# Rather than this
print(5 < x and x < 15)
# You can use this
print(5 < x < 15)
```

Short circuiting

The operators `or` and `and` are said to be **short circuiting** operators, because they only evaluate their second expression if they need to.

To illustrate, the program below does not generate an error, even though the variable `var` has not been defined. That's because Python doesn't even look at that variable. It doesn't need to, because the first `True` is enough to make the whole expression true.

Run PYTHON

```python
print(True or var)
```

Similarly, the program below does not generate an error, even though the variable `var` has not been defined. That's because Python doesn't need to look at it - the first `False` is enough to make the whole expression false.

Run PYTHON

```python
print(False and var)
```

If you swap the order in either case, you get an error:

Run PYTHON

```python
print(var and False)
```

Controlling program flow

In a simple program, Python executes the statements of the program one-by-one, from start to finish. However, you will often want your program to deviate from this:

- **Conditional execution**. You might want to execute a certain statement only if a certain condition is true.
- **Loops**. You might want to loop through a block of code multiple times.
- **Handling exceptions**. You might want to run a block of code in a "cautious" way, so that if an error arises you can deal with it without your whole program stopping.

We will look at these one-by-one in the next few slides.

If statements

You will often want Python to execute a certain statement only if a certain condition is true. For this you can use an `if` statement.

The following program uses an `if` statement:

Run PYTHON

```
1 number = int(input('What is your favourite number? '))
2 if number == 42:
3     print('That is my favourite number too!')
4 print('Good bye')
```

If the user enters the number 42 then Python executes line 3, otherwise it skips line 3 and goes straight to line 4.

Syntax

The syntax for an if statement is:

```
if <expression>:
    <statement(s)>
```

There are two parts to this statement - the part between `if` and `:` is called the **header** of the statement; the rest is called the **body** of the statement. The body of the statement is a **block** of statements.

Notice that the body contains one or more statements (as many as you like), so an `if` statement contains other statements as part of it. Because of this we call it a **compound statement**.

Also notice that the body is **indented**. This is required - if you don't use indentation then Python will issue an error:

Run PYTHON

```
if True:
print('Hello')
```

You can use either the tab character or the space character to create the indentation, but you must use the same character for each line, and the same number of those characters, otherwise Python will issue an error. Each of the following will cause an error:

▸ Run PYTHON

```
if True:
    print('Hello') # Indentation using a tab
    print('Hello') # Indentation using 4 spaces - Error
```

▸ Run PYTHON

```
if True:
    print('Hello') # Indentation using 4 spaces
     print('Hello') # Indentation using 5 spaces - Error
```

It is standard to use 4 spaces for indentation. You can set this to be the default behaviour in the code editor for ed by selecting "Soft tabs" under the settings menu (icon: ⚙) in the top right of an editor window.

The statement block after `if` must contain at least one statement. It is common use the `pass` statement as a placeholder for unfinished code - it is a statement that does nothing:

▸ Run PYTHON

```
if True:
    pass # There must be at least one statement in the body
```

Conditions

Any boolean expression can be used between the `if` and the `:` :

▸ Run PYTHON

```
if True:
    print('The condition is true')
if 2 > 1:
    print('The condition is true')
if 2 > 1 and 2 < 3:
    print('The condition is true')
if 1 > 2 or 2 > 1:
    print('The condition is true')
if 'cat' == 'cat':
    print('The condition is true')
```

Be careful with variables

If you introduce a variable inside the block of an if statement then that variable will only be defined if the block is executed. This might cause you some unexpected errors. For example:

▸ Run PYTHON

```
if False:
    x = 1
print(x) # Error - x does not have a value
```

Since line 2 is not executed the variable `x` does not get assigned a value, so when it is used in line 3 Python issues an error.

Else clauses

You can add an `else` **clause** to an `if` statement, to tell Python what to do if the condition of the `if` statement is false:

▸ Run PYTHON

```python
number = int(input('What is your favourite number? '))
if number == 42:
    print('That is my favourite number too!')
else:
    print('That is not my favourite number.')
```

Elif clauses

You can add `elif` clauses (short for 'else if') to chain together multiple conditions:

▸ Run PYTHON

```python
number = int(input('What is your favourite number? '))
if number == 42:
    print('That is my favourite number too!')
elif number == 21:
    print('That is my second favourite number')
else:
    print('That is neither of my favourite numbers.')
```

Abbreviations

If you only have one statement in an `if` body then you can put it on the same line as the header. The same applies to `elif` and `else`. Note that you still need the colon:

▸ Run PYTHON

```python
number = int(input('What is your favourite number? '))
if number == 42: print('That is my favourite number too!')
elif number == 21: print('That is my second favourite number')
else: print('That is neither of my favourite numbers.')
```

Nesting

Inside the block of an `if` statement you can have other `if` statements. These other `if` statements are said to be **nested**:

Run PYTHON

```python
number = int(input('What is your favourite number? '))
if number > 10:
    print('That is a big number')
    if number > 100: # This if statement is nested
        print('It is bigger than 100')
print('Good bye')
```

Breaking up complex expressions

Do not try to do too much in one go by building overly complex expressions; code should not be concise at the expense of readability. Consider the following two pieces of code:

PYTHON

```python
if (is_admin and not admin_expired) or (is_person and (has_override or s
    call_security()
else:
    activate_launch()
```

PYTHON

```python
if is_admin and not admin_expired:
    call_security()
elif is_person and has_override:
    call_security()
elif is_person and special_override:
    call_security()
else:
    activate_launch()
```

The second piece of code is vastly more clear on the conditions required for a launch.

Ternary expression

Python has an expression whose value depends upon a condition.

```
<expression> if <expression> else <expression>
```

It is called a **ternary** expression, because it combines three expressions into one.

Here is an example of it being used to assign a value to a variable:

Run PYTHON

```python
x = 23
parity = "even" if x % 2 == 0 else "odd"
print(parity)
```

This is equivalent to using the following if statement:

Run PYTHON

```
x = 23
if x % 2 == 0:
    parity = "even"
else:
    parity = "odd"
print(parity)
```

Notice that the ternary expression is an *expression*, not a statement. This means that it returns a value, which you can assign to variables or print:

Run PYTHON

```
x = 23
print("even" if x % 2 == 0 else "odd")
```

An if statement is not an expression, so it doesn't return a value that you can assign to variables or print.:

Run PYTHON

```
x = 23
print(if x % 2 == 0: "even") # Error
```

Be careful when using this ternary operator that your code does not become difficult to read. It is usually best to put parentheses around it.

While statements

Sometimes you might want to repeat a set of statements for as long as a certain condition is true. For this you can use a `while` statement.

Here's a program that uses a `while` statement to print the first 10 positive integers:

▶ Run PYTHON

```
1 n = 1
2 while n <= 10:
3     print(n)
4     n = n + 1
5 print('Finished')
```

When Python gets to line 2 it evaluates the condition after `while`. If the condition is true then it executes the statement block below, in lines 3-4, and then returns to line 2 again. If the condition is false then it skips the block and goes straight to line 5.

You could achieve the same effect by using 10 different `print` statements, but using a `while` statement is more elegant and less repetitive. And if you don't know in advance how many integers to print, for example if you want to ask the user, then it might be impossible to use just `print` statements.

The while block

The `while` block can contain any statement(s) you like, including `if` statements and other `while` statements. For example, here's a program that prints the even numbers between 1 and 10. It uses an `if` statement inside the `while` loop:

▶ Run PYTHON

```
n = 1
while n <= 10:
    if n % 2 == 0:
        print(n)
    n = n + 1
print('Finished')
```

Here is an example of a `while` loop used to iterate through a string, counting the number of times `'e'` occurs in it.

Run PYTHON

```python
string = 'The quick brown fox jumped over the lazy dog'
occurrences = 0
i = 0
while i < len(string):
    if string[i] == 'e':
        occurrences += 1
    i += 1
print("The letter 'e' occurs", occurrences, "times")
```

Continuing

You can use a `continue` statement to skip to the next iteration of a loop:

Run PYTHON

```python
1 i = 0
2 while i < 10:
3     i += 1
4     if i == 5:
5         continue
6     print(i)
7 print('Finished')
```

When the value of `i` gets to 5 the `continue` statement is executed, and Python jumps directly back to line 2 and continues. The number 5 does not get printed, but 6 - 10 do.

Why might you use `continue`? It can help to keep your code from getting to many levels of indentation. We will see examples of this.

Breaking

You can use a `break` statement to break out of a loop entirely:

Run PYTHON

```python
1 i = 0
2 while i < 10:
3     i += 1
4     if i == 5:
5         break
6     print(i)
7 print('Finished')
```

When the value of `i` gets to 5 the `break` statement is executed, and Python jumps directly to line 7. The number 5 does not get printed, and nor do 6 - 10.

Note that if the loop is nested inside another loop, the `break` statement terminates only the inner loop.

Keeping a program running

When you run the program below is stops after it gets and prints a name. To run it again you have to click 'Run' again:

▸ Run PYTHON

```
name = input('What is your name? ')
print('Hello', name)
```

It can be convenient to have the program keep running - getting it to start again automatically after it does its thing. You can get it to do this by adding a `while` loop, with a condition that always evaluates to true:

▸ Run PYTHON

```
while True:
    name = input('What is your name? ')
    print('Hello', name)
```

Now the program will keep running, until you click 'Stop'.

If you want to get a bit fancier, you could get your program to stop when the user enters a certain value, such as 'q'. Remember to let the user know that they can do this:

▸ Run PYTHON

```
while True:
    name = input('What is your name? (Enter q to quit) ')
    if name == 'q':
        break
    print('Hello', name)
```

Handling exceptions

If your code interacts with the outside world, you may encounter unexpected circumstances. Perhaps a file that you are trying to open doesn't exist, or when you try to save the user's data you find that the disk is full, or perhaps the user enters a non integer value when you are expecting an integer, or maybe you divided by zero.

Exceptions are a mechanism for dealing with the unexpected.

The approach in Python is known as **structured exception handling**. This means that if Python encounters an exception in a block of code, it will search for an exception handler enclosing that block. If none exists, it will look for an exception handler enclosing that outer block, and so on.

It is important to understand that exception handling should be used to handle exceptions that are recoverable. If your program cannot handle the exception or recover from it, it should not even try. Instead it should let the exception terminate your program so the user can deal with it instead. For many cases of exceptions, there is no remedy.

Try and except

You can handle exceptions by using a `try` statement.

Consider the following program, which asks the user to enter a number and then returns the square of that number:

▸ Run PYTHON

```python
num = input("Please enter a number: ")
result = float(num) ** 2
print("The square of the number is", result)
```

If the user enters a string of letters instead of a number then an error occurs and the program very ungracefully ends.

Python complained with a `ValueError`. To handle this, we'd write a `try ...except ...` block around the code that could potentially raise an exception.

▸ Run PYTHON

```python
num = input("Please enter a number: ")
try:
    result = float(num) ** 2
    print("The square of the number is", result)
except:
    print("You did not enter a number.")
```

In the above code, when control reaches line 2, an exception is raised. Python finds the nearest exception handler and moves control to line 3, which proceeds into the block at line 4. If there was no exception on line 2, then the code in the `except` block would not be executed.

If you don't want to do anything with the exception then you can use the `pass` statement:

▸ Run PYTHON

```python
num = input("Please enter a number: ")
try:
    result = float(num) ** 2
    print("The square of the number is", result)
except:
    pass
```

When an exception occurs, there is information about the exception contained in a special Exception object. You can assign that object to a variable and then display that information to the user:

▸ Run PYTHON

```python
try:
    int('string')
except Exception as err:
    print('Exeption: ', err)
```

Else

If you have some code that you want to execute only if the `try` block succeeds then you can add an `else` clause:

▸ Run PYTHON

```python
x = 'Hello'
try:
    x = int(x)
except:
    print('The conversion was not successful.')
else:
    print('The conversion was successful.')
```

Working with files

So far we have seen programs that interact with the external environment via the `input` and `print` functions. Another way they can interact is by reading from and writing to files. This is very simple in Python, which is why Python is a popular tool for working with files.

This is what you will typically want to do:

- Open the file
- Read from the file, or write to the file
- Close the file

Opening a file

To work with a file you must first open it. You do so using the `open` function:

▸ Run PYTHON

```python
f = open('MyData', 'w')
```

The function expects a **file** as the first argument. This can be given as an absolute or relative filename. If given as a relative filename, it is relative to the directory that Python was executed from - in Ed this is always the same directory as the program.

You can also supply a **mode** as the second argument. This indicates whether the file is to be opened for reading (i.e. input) or writing (i.e. output). If you don't specify the mode, Python assumes that you want to open the file for reading.

The available options for the mode are:

- `'r'` - Open the file for reading. `open` will throw an exception if the file does not exist.
- `'w'` - Open the file for writing. If the file exists, the contents are completely overwritten. If the file does not exist, it will be created.
- `'x'` - Open the file for writing. If the file already exists this will throw an exception. If the file does not exist, it will be created.
- `'a'` - Open the file for appending. The file is opened at the end and any writes to the file will append to the end. If the file does not exist it is created. This option is useful for adding information to a file - for example a log file.

The `open` function returns a file object (also called a **file handle**) that represents the file that you have opened. It is this object that allows you to perform operations on the underlying file itself.

Reading from and writing to a file

The `read` method of a file object returns a string that contains the entire contents of the file. The `write` method takes a string and adds it to the file.

Run PYTHON

```python
# Open a file for writing
file = open('myfile', 'w')

# Write to the file
# If you want a newline anywhere you have to add it, using \n
file.write('Line 1: Some text.\n')

# Write some more to the file
file.write('Line 2: Some more text.')

# Now open the file for reading
file = open('myfile', 'r')

# Print the contents
```

Closing a file

You can close a file by using its `close` method. It is important to close a file after you are done working with it, to free up resources back to the system. A program can only have a limited number of files open while it is running. If a running program reaches this limit, it will receive a "Too many open files" error when it attempts to open more files.

So the program above should look like this instead:

Run PYTHON

```python
file = open('myfile', 'w')
file.write('Line 1: Some text.\n')
file.write('Line 2: Some more text.')
file.close()

file = open('myfile', 'r')
print(file.read())
file.close()
```

Using a `with` block

It's a good idea to work with a file inside a `with` statement. This ensures that the file is always closed after use, even if an error occurs inside your program. As soon as control exits the `with` statement the file will be automatically closed.

Here is the previous example written with `with` statements:

Run

PYTHON

```python
# Open a file for writing
# The file is automatically closed after the with statement
with open('myfile', 'w') as file:
    file.write('Line 1: Some text.\n')
    file.write('Line 2: Some more text.')

# Open the file again for reading
# The file is automatically closed after the with statement
with open('myfile', 'r') as file:
    print(file.read())
```

Further reading

You might find the following helpful:

- Chapters 2, 5, 7 and 10 of Python Crash Course
- The Python Tutorial at w3schools.com