MATH5306M Introduction to Programming

Reference of Python commands and syntax

Christopher F. Tedd

2024/25

Contents

Basic operations

Variable assignment

We can store data values so that we can refer to them elsewhere in our programme using a *variable assignment* with the = symbol. We write our chosen *variable name*, then the = symbol, then the data we wish to keep.

```
number = 4
message = "Hello, world!"
decision = True
```

Whenever we use the variable name elsewhere in our code, Python will use the value of the data that we have stored in the variable.

Basic operations on data

Arithmetic

We can perform basic arithmetical operations on numerical data

Arithmetical operation	Python operator
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Exponentiation	`**`
Inetger division (whole number after division)	`//`

```
total = 4 + 17
fraction = a / b
```

If we want to change a variable by applying some arithmetical operation to its current value, we can use an *assignment operator*: this is a combination of the above operator symbol and the `=` symbol; e.g.

```
total += 17
```

will add `17` to the current value of the variable `total` (and store it in that same variable).

Text

Some of these operators behave differently if we use them on data in a text form. The `+` operator *concatenates* two strings of text, i.e. joins then together into a single piece of text data. The `*` operator acts as a *repetition* operator, repeating the text by the given number of times.

Operation	Python operator	Example
Concatenation	`+`	`"abcde" + "fghij"` gives `"abcdefghij"`
Reptition	`*`	`"a" * 10` gives `"aaaaaaaaaa"`

Conditions and logical operators

In getting a programme to function correctly on various different possible data, it's very important to test the data for certain properties and record whether a condition – or several conditions in combination – is `True` or `False`. Firstly we have Python operators for performing comparisons between two data values:

Equal to	`==`
Not equal to	`!=`
Greater than	`>`
Less than	`<`
Greater than or equal to	`>=`
Less than or equal to	`<=`

Note that the last four comparisions also work on text, comparing the pieces of text against alphabetical order.

To combine information about whether several conditions have been met or not, we need to use logical operators:

Logical operator	Python operator
AND	`&`
OR (inclusive 'or')	`|`
XOR (exclusive 'or')	`^`
NOT	`!`

Data types

As has already been made clear by what we have discussed above, we can have different *kinds* of data – sometimes we will have numerical data, sometimes it will be text. Furthermore, sometimes it will be information about whether something is true or false. Python has a system of *data types* that allows it to distinguish between these different kinds of data, to ensure that it performs the appropriate operation on the data.

Data type	Description
`int`	A whole number. If you enter a number without any decimal point in a variable assignment, the variable will be stored as `int` type. Furthermore, if you use any of the following operators on two data values which are both `int`s, the result will also be of type `int`: `+`, `-`, `*`, `//`.
`float`	A decimal number (stored in a '*floating point*' format). If you are entering a whole number but you want it to be of the `float` type, you can insert a decimal point at the end of the number, e.g. `a = 4.` The outcome of any arithmetic operation involving at least one `float`, along with any calculation using the `/` operator is always a `float`, even if the answer is a whole number.
`str`	A sequence of keyboard characters e.g. letters, numbers, symbols. In a variable assignment, to create a string, you enclose the text you want to store in the variable between either single-quote marks `'` or double-quote marks `"`. You can also create multi-line strings by using three opening and closing quote marks (either single or double quotes).
`bool`	A logical value which is either `True` or `False`. The comparison operators listed above produce data of the `bool` type.

Additional string functionality

Format strings are a very useful Python functionality enabling text and Python code to be combined in a neat and readable manner. By putting an `f` before the opening quote mark (whether a single- or multi-line string) you create a format string. Inside the following text, anything enclosed within curly brackets { } will be evaluated as python code, allowing you to incorporate variable names, inbuilt functions and so on. For example, if you have a value stored as a `float`, it

might make sense to round to a certain number of decimal places when printing a message to the user, without rounding the stored value itself. You could then use a format string:

```
print(f"The value is {round(my_value, 5)} to 5 d.p.")
```

Inbuilt functions

There are certain keywords which if we type them into a Python programme, Python will perform a builtin operation:

Function	Description
`print(`*`[string]`*`)`	Outputs the text contained in *`[string]`*. For example, `print("Hello, world!")` displays the text `Hello, world!`
`input()`	Takes text typed in by the user, e.g. for use in a variable assignment. You can pass an optional string to the function which will be displayed as a prompt. The data returned by `input()` will always be of `str` type.
`round(`*`[number]`*`)`	Returns the value of *`[number]`* rounded to the nearest whole number, rounding to the nearest *even* number when the data is an exact half *(Bankers' rounding)*. Can be used to obtain an `int` from data currently of `float` type, e.g. `round(10.5)` gives the `int` `10`.
`round(`*`[number], [places]`*`)`	Returns the value of *`[number]`* rounded to *`[places]`* decimal places, e.g., `round(1.234, 2)` returns `1.23`. Always returns a `float` even if *`[places]`* is `0`.
`type(`*`[data]`*`)`	Returns the type of the data given as argument
`range(`*`[number]`*`)`	Can be used, for exampe in a `for` loop, to successively produce the values `0` to `n-1`
`range(`*`[start, end]`*`)`	Will produce the values from `start` up to `end`−1.

Typecasting functions

Often we need to modify the type that our data is – for example, if we are asking for a number as an input, as the `input()` function only returns a `str`. Each type has a corresponding *typecasting* function that tries to convert from other data types to the given type.

Function	Notes
`int(`*`[data]`*`)`	If *`[data]`* is a `float`, then `int()` returns the whole-number part of the input – this is *truncation*. If the input is a `str` then `int` will only succeed if the string contains only numerical characters with optionally a sign (`'+'` or `'-'`) as the first character – it does not accept strings containing a decimal point.
`float(`*`[data]`*`)`	Can convert an `int` or a `str` to a `float`.
`str(`*`[data]`*`)`	Converts most Python data to a string containing the usual Python representation of that data – e.g. if you apply `str` to a float-type variable equal to `1`, it will return the string `"1.0"`.

Control flow

One of the fundamental things we require a computer programme to do is to respond differently based on data or input it is given. Getting a programme to run different lines of code other than just running from start to finish is known as *control flow*. In Python, control flow consists of a *keyword*, often followed by a condition or some additional information, then a `:` symbol. Python then identifies the code relating to the control-flow statement by *indentation level*, that is, code following the `:` which is indented further than the keyword is identified as the code corresponding to that keyword.

The most straightforward form of control flow is when you want to run one set of code if a condition is true, and possibly some different code when the condition is false (where we may then wish to test against some other conditions). In Python this uses the `if`, `elif` and `else` keywords.

```
if my_value < 10:
    print("Yes! It's less than 10")
elif my_value == 10:
    print("It's not strictly less than 10, but I will accept it")
else:
    print("The value is too large")
```

Note that `elif` and `else` statements are optional, and that you can have several `elif` statements testing different conditions.

Another fundamental control flow task we very often want to use in a programme is to run identical lines of code several times but where some of the values have changed, such as running code on each item on a list in turn.

We can use the `for` keyword to run a loop with an *index* which keeps track of which repetition of the loop we are in. The `range()` will return, successively, integer values from `0` up to one less than the number we gave as argument. For example, `range(5)` will successively give the values `0`, `1`, `2`, `3` and `4`. The following code

```
for i in range(5):
    print(i)
```

will print the numbers from 0 to 4.

There is another loop command, `while`, which takes a condition, and at the start of each run of the loop, checks whether the condition is true. If it is, the

block of code corresponding to the `while` keyword is run (i.e. the code after the `:` symbol which is indented more than the `while` keyword) and the programme returns to the start of the block to check the condition again. If the condition is false, the programme moves past the `while` block onto the subsequent lines of code. This kind of loop must be used carefully to ensure that, in running the associated block of code, something will eventually change which will make the condition false, as otherwise you will end up with an infinite loop. For example:

```
i = 0
while i < 5:
    print(i)
    i = i + 1
```

will again print the numbers from 0 to 4, and then exit. In such a case, a `for` loop would have been the more appropriate choice.