Programming Concepts

Ben Wright

16th October 2019

Who is this course for?

- People with no previous programming experience
- People who've 'learned by doing' as part of their work or study but want a better grounding
- People who want a refresher after not programming for a while

What is programming?

- Creating a set of instructions for a computer to follow
- Three main challenges:
  - Getting it to do anything at all
  - Getting it to do what you told it to do
  - Getting it to do what you actually wanted

What makes computers tick?

- Computers use **machine code**
- When this code is **run**, it takes **input** and produces **output**
- Different programming languages are different ways of specifying instructions, which then get turned into machine code

Compiled versus Interpreted

- For **compiled** languages, after your code is written it is ‘compiled’ into machine code by a special program
  - Examples: C, C++, Fortran
- It needs to be compiled separately for different operating systems, different processors, etc.

Compiled versus Interpreted

- For **interpreted** languages, compilation is performed behind the scenes when you try to run the code
  - Examples: Python, R, bash

Compiled versus Interpreted

- Compiled languages let you catch some errors before you try running the code
- Compiled languages are, usually, faster to run
- Interpreted languages are more **portable**
- Interpreted languages have an interactive **interpreter** you can try your code out in

Running code

- Anything you type into an interpreter is carried out immediately
  - e.g. in R, or the python interpreter
- **Scripts** are simple text files containing code that an interpreter can process
  - e.g. Bash scripts, R scripts, most python code

Running code

- Compiled languages require you to compile the code using a program that comes with the language, then run it separately
  - Code is often spread out over multiple files, even for the same program

Pseudocode

- **Pseudocode** is something halfway between how you'd describe the program, and the code itself
- It's used as a way to work out what you want your code to do before you try to write it

Pseudocode

- There is no formal definition of what pseudocode looks like
- It should be easier to understand than the code itself, but make explicit what steps the program will perform

Pseudocode Example

```
Set C to 0
For each value A in a list of values:
   If A is 0, set C to 0
   Otherwise, set C to C+1
Print C
```

- What does this do?

Keywords

- Most languages have a set of **reserved keywords** that are the nuts and bolts of the language
- These are always taken to be instructions for the language and if you try to use them for something else you will generally get an error
  - Examples: if, for, import, return, elif, fi

Symbols

- Each language uses symbols to represent mathematical or logical operations and these are fairly consistent between languages

Symbols

Language	Python	R	Java
Addition	+	+	+
Subtraction	-	-	-
Multiplication	*	*	*
Division	/	/	/
Power	**	^	Math.pow()
Join strings	+	paste()	+
And	and	& or &&	& or &&
Test equality	==	==	== or equals()
Modulo	%	%%	%

Comments

- Almost all languages have a way for you to add text to your code that is ignored when the code is run, called **comments**
- You should add comments to your code to make it easier to understand
- Do not just say what the code is doing, say **why**

Datatypes

- Data is stored in **variables**, which you give names to
  - There are usually generous rules as to how you can name them
  - Reserved keywords are forbidden
- Different types of data are often handled and/or stored differently

Numeric Datatypes

- Numbers are generally divided into **integers** and **floating point** numbers
  - Integers are perfectly precise
  - Floating point numbers have a wider range of values, and support decimals
- Some languages also have different sizes of numeric value, to store larger/more precise values

Datatypes

- **Boolean**: a True or False value.
- **Character**: a single typed character
  - Examples: “A”, “a”, “@” or “4”
- **String**: a sequence of characters
- Other data types such as **objects**, and specific data structures which are covered shortly

Datatype declaration

- In some languages, you have to explicitly **declare** the data types for each variable you create
  - Examples: C, Java
- In others, you do not
  - Examples: R, python

Datatype casting

- **Casting** means converting one data type to another
  - e.g. turning the string “413” into the integer 413
  - e.g. turning the integer 413 into the floating point number 413.0
- In some languages you have to explicitly tell the programme to make the conversion

Datatype casting

- Some languages will do their best to convert automatically, as needed
- Those languages also allow you specify explicitly
- Converting to and from strings is often done somewhat differently

Data structures

- **Data structures** store multiple bits of data in one place, in a logical way
- They are mostly used the same way as other data types, and are stored in variables
- In languages that use objects, these structures are themselves objects

Arrays / vectors

- A linear sequence of data, of a known length and all of the same data type, is called an **array**
- You can directly access elements of an array by their **index** (usually using square brackets)
- R is a special case in which nearly everything is a **vector** (i.e. an array), even if only of length 1

Maps / dictionaries / lists

- A similar structure is the **map**, where you can use anything as the index, not just integers
- The elements are usually unordered
- They are also called **associative arrays**, **dictionaries** (in python) and **lists** (in R)

Other data structures

- Linked lists
- Queues
- Binary trees
- Records / tuples / structs

Code blocks

- A **block** is a series of instructions that is meant to be performed together in sequence
- Some code structures use blocks to define their beginning and end
- Languages differ in how you define blocks, but it’s usual practice to indent them from the surrounding code

Code blocks

```
x = y * 2
if x >= 10:
   is_double_digit = true
   display_mode = “special”
z = x + y * 3
```

Conditional blocks

- Perform a test, and then run a code block only if that test was true (usually '**if**')
  - Can be followed up by a different code block to perform if the test was false (usually '**else**')
  - Most languages have a convenient way of combining these so you can have a list of such tests ('**else if**' or '**elif**')

Conditional blocks

```
if x > 10:
   do something
elif x > 5:
   do something else
else:
   do a third thing
```

Loops

- **Loops** let you perform the same steps over and over again
- Usually with slight variation in the data being used

Loops

- Three kinds of loop:
  - You know in advance exactly how many times the block needs to run (**for loop**)
  - You test a condition before each iteration of the loop (**while loop**)
  - You test a condition at the end of each loop (**until loop**)

Loops

```
int y=0;
while(x > 0) {
   y = y + x * x;
   x = x - 1;
}
```

Functions

- **Functions** are parts of the language that perform operations more complicated than simple arithmetic
- If you need to do do something in more than one place in your code, it is better to make it a function than to copy/paste it

Functions

- Making use of a function is **calling** a function
- Functions have three main parts:
  - **Arguments**
  - **Code**
  - **Return value**

Function arguments

- The arguments are given to the function in order to provide detail on what it needs to do
- When you use a function, you need to provide these arguments

Function code

- If you are using a function already defined in the language, you do not need to know about the code inside it
- You can, however, write your own functions
  - You write the code block that takes the function arguments and (optionally) returns a value

Function return value

- Some, but not all, functions return a value when you use them.
- This is usually the result of a calculation, but it can also be other information about what the function did
- You can call a function anywhere you would use a value, as long as the function returns a value of the right type

Function side effects

- Sometimes, a function can change a value
  - Either one of the arguments, or a value stored somewhere else
- This is usually called a **side-effect** of the function
- Not realising that a function has side effects is a common source of coding errors

Functions

```
triangular <- function(x){
   x * (x+1) / 2
}
```

Scope

- Variables are only visible to certain parts of your code, according to rules in the language
- Languages typically have a means to specify this **scope** explicitly to make sure you can use variables where you need them
- For example, variables used solely inside a function disappear once the function returns, in most languages

Libraries

- The code you will write will often use functions in pre-existing code
- The places where this handy code can be found are called **libraries**, **modules**, **packages** or something similar.
- There are several different ways to access these libraries

Libraries

- Some functions are in-built parts of the language
  - You can just use these functions without taking any extra steps
- Some are standard libraries that are always available, but you have to say that you are going to use them
  - How this is done varies by languages, but is often called an **import**.

Libraries

- Third party libraries are downloaded separately
  - Sometimes you can make them available the same way you would for standard libraries
  - Often useful libraries are installed alongside the standard libraries anyway, or are already installed on the system you are using
- Compiled languages frequently have a more complicated and error-prone mechanism for linking to libraries

Errors

- By default, when a serious error occurs the programme stops – a **crash**
- If you are lucky, you will get an explanation of the error that says:
  - What the error was
  - Where in your code the error happened

Errors

- Silent errors, which don’t stop the programme but lead to incorrect results, are far more serious and harder to detect
  - Never take on faith that your code is working properly just because it didn’t crash

Best practises and style

- These are recommendations, usually specific to a language, to make your code ‘good’
- Having code that works properly is only half the battle
- You need to make your code as easy to understand as possible

Where to find out more

- There's a longer, more thorough 'Programming: Concepts' course offered through central IT:
  - http://portfolio.it.ox.ac.uk/resource/course-pack/programming-concepts
- TED-Ed is running a beginner's programming strand:
  - https://www.youtube.com/watch?v=KFVdHDMcepw
- Rosetta Code provides side-by-side comparisons of the same code in different languages:
  - http://rosettacode.org/