Python Notes

Table of Contents

Background and setup....2
  Learning Python....2
  Setup....2
  Three ways to use Python....2
Basic ideas....5
  Input, process, output....5
  Assignment operators....7
  Syntax....8
  Loops....9
  for loops with range....11
  Data structures – lists....11
  Maps – dictionaries....13
  Files....13
  Exceptions....14
Structured programming....16
  Python functions....17
  Function parameters....18
  Function returns....18
  Scope....19
  Pass by value....20
  Built-in functions....21
  Recursive functions....21
Object-oriented programming....24
  Objects....24
  Classes....24
  Methods....25
  Constructors....25
  Special method names....26
  Per class fields....27
  Inheritance....28
  Class hierarchies....29
  Over-riding methods....29
  Abstract base classes....30
Not So Basic Ideas....31
  All data are objects....31
  All named code is data....31
  Useful built-in functions....31
  The object class....31
  Special name attributes....32
  Operator overloading....32
  Encapsulation and access control....33
  Bytecode....35
Glossary....38

copyright © W W Milner 2021

Background and setup

This is a set of notes introducing basic programming in Python.

It does not cover all of Python. It is not a reference work. References are very big and need to be kept up to date, and so change with every new version. You can find that, and other useful things, at

https://www.python.org/doc/

Learning Python

Python is a programming language. There are only 28 words in the language, so it might seem quick and easy to learn. But the words use a lot of ideas which take a lot of work to understand.

This text is just a start.

It does not cover data structures and algorithms, which are ideas which are the basis of how you design programs – how you work out how a program will work. Once you can write basic Python, you need to learn those topics.

Setup

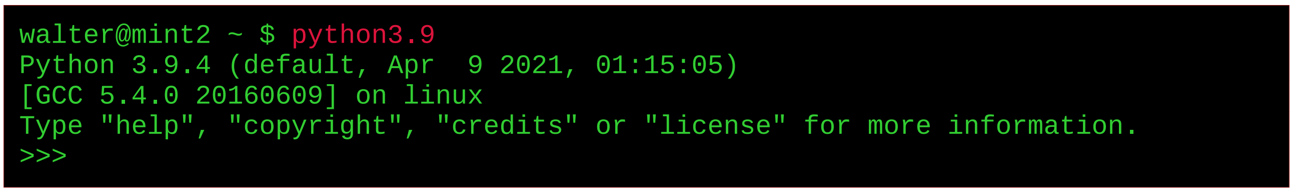
You may be using Python on a college computer. If so you can ignore this section and just used the installed version. Read the documentation supplied by your college.

To set up Python on your own device, that depends on which OS you are using, and which version of Python. The best thing is to Google it.

Three ways to use Python

1 Interactively

Start Python. Screenshots show this for the current version at June 2021. The version does not matter much:



```
walter@mint2 ~ $ python3.9
Python 3.9.4 (default, Apr  9 2021, 01:15:05)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

What you type is in red.

This is the interpreter running. In interactive mode, we can type in one Python instruction at a time. Sometimes a result will appear. For example

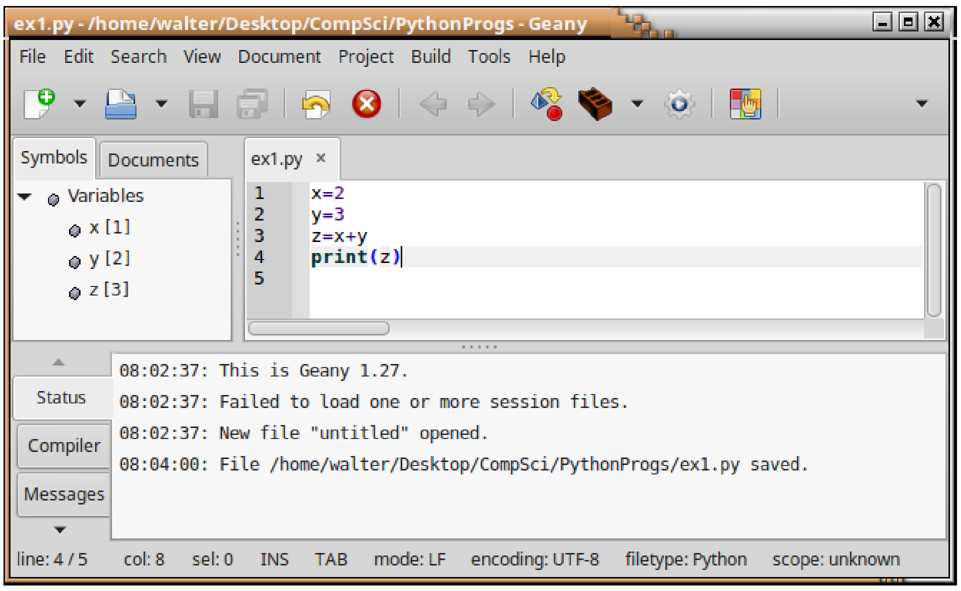
```
walter@mint2 ~ $ python3.9
Python 3.9.4 (default, Apr  9 2021, 01:15:05)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x=2
>>> y=3
>>> z=x+y
>>> print(z)
5
>>> quit()
walter@mint2 ~ $
```

quit() stops the interpreter running.

2 Executing a script

Interactive mode is simple and easy. But it is limited. Another way is to put the Python instructions into a script, using a text editor, and storing the script in a file. Then we tell the interpreter to run the script.

A text editor is a program which lets you edit text. Unlike a word processor, we have no formatting, such as fonts or colours. In Windows, notepad is a text editor. In Linux, geany is a text editor. For example:


ex1.py - /home/walter/Desktop/CompSci/PythonProgs - Geany
File Edit Search View Document Project Build Tools Help
Symbols
Documents
Variables
x [1]
y [2]
z [3]
ex1.py
Status
Compiler
Messages

```
x=2
y=3
z=x+y
print(z)
```

```
08:02:37: This is Geany 1.27.
08:02:37: Failed to load one or more session files.
08:02:37: New file "untitled" opened.
08:04:00: File /home/walter/Desktop/CompSci/PythonProgs/ex1.py saved.
```

line: 4 / 5   col: 8   sel: 0   INS   TAB   mode: LF   encoding: UTF-8   filetype: Python   scope: unknown

This shows using geany. We have typed in 4 Python instructions, then saved them in a file named ex1.py. Python scripts usually end .py.

Then at the command line we navigate to where we saved ex1.py, and run Python, giving the script name ex1.py on the command line:

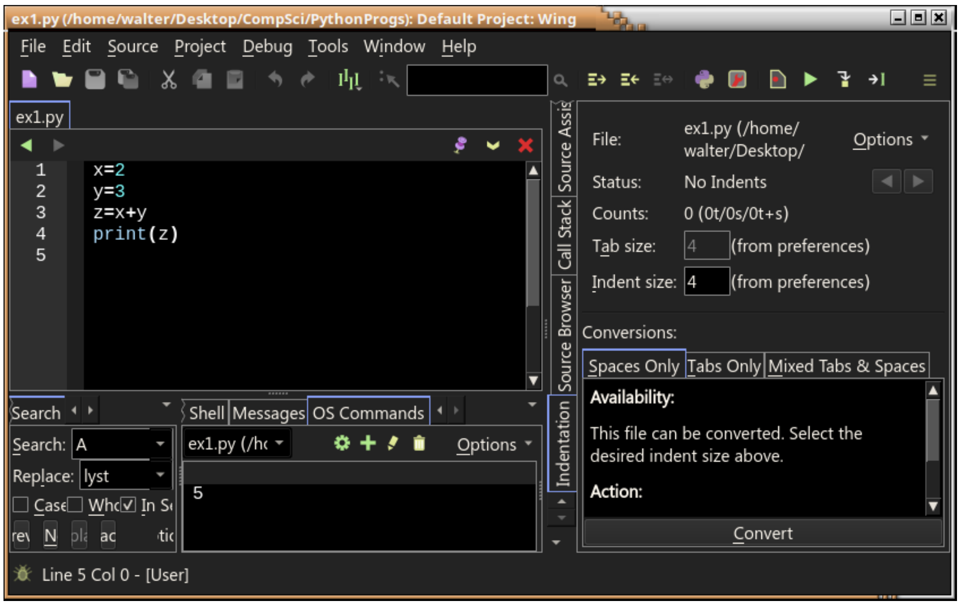
```
walter@mint2 ~ $ cd Desktop
walter@mint2 ~/Desktop $ cd CompSci
walter@mint2 ~/Desktop/CompSci $ cd PythonProgs
walter@mint2 ~/Desktop/CompSci/PythonProgs $ python3.9 ex1.py
5
walter@mint2 ~/Desktop/CompSci/PythonProgs $
```

There are many advantages to doing it this way. The script is a *stored program* – a fundamental Computer Science idea.

3 In an IDE

An IDE is an integrated development environment. This is an application which gives access to an interpreter or compiler, and editor, a debugger and other tools,conveniently in a single environment. They can usually be configured to use diffeernt programming languages.

Wing is an IDE designed for Python:



ex1.py (/home/walter/Desktop/CompSci/PythonProgs): Default Project: Wing

```
x=2
y=3
z=x+y
print(z)
```

Output: 5

File: ex1.py (/home/walter/Desktop/
Status: No Indents
Counts: 0 (0t/0s/0t+s)
Tab size: 4 (from preferences)
Indent size: 4 (from preferences)

Conversions: Spaces Only | Tabs Only | Mixed Tabs & Spaces

Availability: This file can be converted. Select the desired indent size above.

Action:

Search: A
Replace: lyst

Line 5 Col 0 - [User]

File Edit Source Project Debug Tools Window Help
Shell Messages OS Commands
Options
Convert
Source Browser Call Stack Source Assist Indentation

We run the script by saying Debug.. Execute current file.

IDEs offer a lot of tools, so can be confusing for beginners.

Basic ideas

Input, process, output

Suppose we have an eCommerce website. We work out the price of each product in a way to encourage sales, as follows:

> Find the cost of the product to us, and the quantity bought
>
> If the quantity is less than 100, the price is 50% more than the cost. If the quantity is more than 100, the price is only 10% more than the cost.

We want a Python program to work this out for us. Here it is:

```
# input data
cost=input("Enter cost")
quantity=input("Enter quantity")
# calculate price
cost=int(cost) # change string type to int
quantity=int(quantity)
if quantity<100:
    price=cost*1.5*quantity
else:
    price=cost*1.1*quantity
# output result
print(price)
```

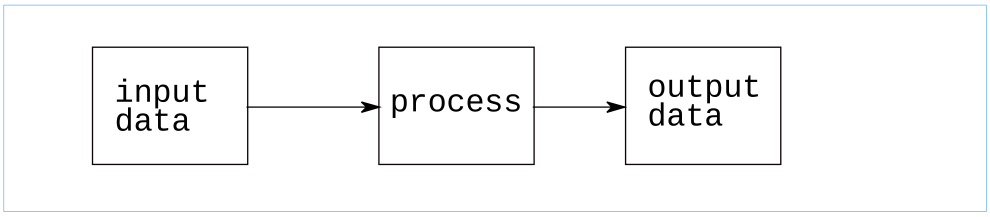
Two sample runs are:

```
Enter cost10
Enter quantity10
150.0
```

and

```
Enter cost10
Enter quantity100
1100.0
```

This shows 3 basic ideas in software:



Here *input* means input of data values into the program. This might be any type of input. In this case it is text typed in from the keyboard, but it might be a mouse click, a packet from a network interface, something from a file, a reading from a temperature sensor, or any other kind of input. This is *runtime input* – happening when the program executes. This is different from *compile-time input* – when we are writing the program.

Process means changing the data, by program code – any kind of programming, any kind of change

Output means sending data out of the program code. In this example it appears on the screen as text. It might be output as a graphics object, or sound, or output to a file, or sent over the network, and so on.

In Python we use input to do input, and print to do output.

The program shows several other computer science ideas:

Variables. cost quantity and price are variables – changing values with names.

Comments. These start with a #, like

```
# input data
```

These just explain what is happening. The system ignores them.

Type. There are different types of data, such as whole numbers (integers, ints), numbers with decimal parts (floating point numbers, float), strings of characters, and others. Here input gives us a string type.

```
 cost=int(cost)
```

is a *type-cast*. This changes one type (string) into another (int).

Statements. A program is a sequence of instructions. Each instruction is one statement, on one line.

Assignment statements. One type of statement is an assignment, to assign a value to a variable- such as

```
    price=cost*1.5*quantity
```

Expressions. These are things for the computer to calculate, such as

```
cost*1.5*quantity
```

We often assign a calculated expression to a variable.

Conditional statements. These start with ‘if’. These have a condition, and the following statements are only execute dif the condtion is true. For example

```
if quantity<100:
```

means we only do what comes next if the quantity is less than 100.

Indentation. This means setting code in from the left hand edge. For example

```
if quantity<100:
    price=cost*1.5*quantity
else:
    price=cost*1.1*quantity
```

the 2 statements in italics are indented. This shows what the if and else apply to. Python is unusual in that indentation really matters. Check the colon : after the if and else

Assignment operators

We often want to increase a variable by 1. That is called incrementing the variable. We can do this for example by saying

```
x = x + 1
```

This means take the value of x, add 1, and store it as the new value of x.

But there is another way:

```
x += 1
```

which does the same thing, but is better, because

- It is probably faster
- It is more readable – its meaning or purpose is clearer.

This is also possible with other operators. For example

```
x += 3# add 3 to x
x -= 2# subtract 2 from x
y *= 4# multiply y by 4
b /= 10  # divide b by 10
```

Syntax

Syntax is another word for *grammar*. Syntax means the rules of the language. Python, like other programming languages, is a formal language. That means it has a set of syntax rules, and if those are broken, the program is invalid.

For example

```
x = 3
y = 2
x + y = z
```

When we try to run this, we get:

```
  File "/home/walter/Desktop/CompSci/PythonProgs/test.py", line 3
    x + y = z
    ^
SyntaxError: can't assign to operator
Process terminated with an exit code of 1
```

This is an error message. It tries to tell us what the error is - can't assign to operator. (The operator is +).

And it tells us where it is:

```
line 3
    x + y = z
    ^
```

We cannot tell it what the value of an expression (x+y) is. It works out the value of an expression for itself. It makes no sense to tell it the value of x+y is z. Maybe we meant:

```
x = 3
y = 2
z = x + y
```

That means work out x+y (get 5) then assign that value to z.

Any program with one or more syntax errors is invalid and will not run. It must be debugged – remove the syntax errors. Read the error messages – they are good clues.

The full grammar of Python is here :
https://docs.python.org/3/reference/grammar.html

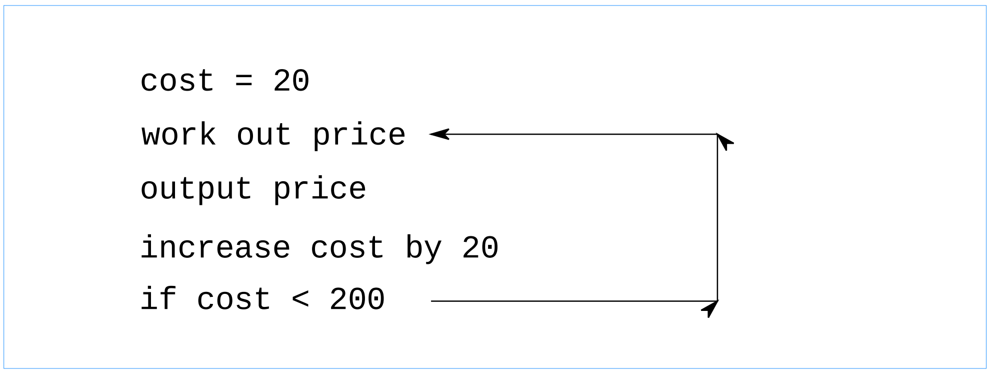
Loops

Suppose on our eCommerce site, where we work out the price of each product in a way to encourage sales, as follows:

> Find the cost of the product to us, and the quantity bought
>
> If the quantity is less than 100, the price is 50% more than the cost. If the quantity is more than 100, the price is only 10% more than the cost.

But we do not want to keep doing this for lots of different costs. We want the program to work this out for a whole set of different costs. In other words, we want to repeat the code. This is called looping, or *iteration*.

The plan is:



```
cost = 20
work out price
output price
increase cost by 20
if cost < 200
```

In Python we can say this using a ‘while loop’:

```
cost=20

while cost<200:
    quantity=50
    if quantity<100:
        price=cost*1.5*quantity
    else:
        price=cost*1.1*quantity
    print("Price of ", quantity, ' at cost ', cost, ' = ', price)
    cost+=20
```

This uses the code from the last section, but places it in a loop.

Check the : after the while statement.

Everything from

```
    quantity=50
```

to

copyright © W W Milner 2021

```
    cost+=20
```

is indented, so this is the *loop body* - the code which is repeated.

Inside the loop there is an if. so

```
        price=cost*1.5*quantity
```

is indented twice.

The output is:

```
Price of  50  at cost  20  =  1500.0
Price of  50  at cost  40  =  3000.0
Price of  50  at cost  60  =  4500.0
Price of  50  at cost  80  =  6000.0
Price of  50  at cost  100  =  7500.0
Price of  50  at cost  120  =  9000.0
Price of  50  at cost  140  =  10500.0
Price of  50  at cost  160  =  12000.0
Price of  50  at cost  180  =  13500.0
```

But this only works it out for a quantity of 50. To be useful we need different costs and prices. We can do this by using what we have, and putting in another loop, which changes the quantity:

```
quantity = 50
while quantity<200:
    cost=50
    while cost<200:
        if quantity<100:
            price=cost*1.5*quantity
        else:
            price=cost*1.2*quantity
        print("Price of ", quantity, ' at cost ', cost, ' = ', price)
        cost+=50
    quantity+=50
```

The output is:

```
Price of  50  at cost  50  =  3750.0
Price of  50  at cost  100  =  7500.0
Price of  50  at cost  150  =  11250.0
Price of  100  at cost  50  =  6000.0
Price of  100  at cost  100  =  12000.0
Price of  100  at cost  150  =  18000.0
Price of  150  at cost  50  =  9000.0
Price of  150  at cost  100  =  18000.0
Price of  150  at cost  150  =  27000.0
```

Red is the first time round the outer loop. Green the second, and blue the third.

These are called *nested loops.*

copyright © W W Milner 2021

for loops with range

In Python there is another kind of loop syntax – a for loop using range, like this

```
for i in range(1,5):
    print(i)
```

Output:

```
1
2
3
4
```

So range(a,b) goes a, a+1, a+2 and so on up to but not including, b.

range can be used in many situations, but is most common in a for loop like this example.

Data structures – lists

We often want to deal with a set of values, not just single items. Examples might be all the students in a class, or all the pixels in an image, or all the purchases on an eCommerce site last week.

We put the values together into a single *data structure*.

Different data structures can be used. Python has a very useful built-in structure, called a list. Python lists are enclosed in [square brackets].

The things in a data structure are called elements.

We can access a list element using an index, which is simply an integer, with the first element having index 0, the next index 1, and so on.

Suppose we have a college management system, and we store the marks of students in a class. We can hold them in a list:

```
marks=[34,45,72,19,23,78,42]
print(marks[0])     # 34 : index 0 = first element
print(marks[1])     # 45
print(marks[2])     # 72
```

We can iterate through a list like this:

```
marks=[34,45,72,19,23,78,42]
index=0
```

```
while index<6:
    print(marks[index])
    index+=1
```

The 6 here is the length of the list. It is better to use the len function for this:

```
marks=[34,45,72,19,23,78,42]
index=0
while index<len(marks):
    print(marks[index])
    index+=1
```

because

- The reader does not have to work out why we are using 6 – it is more readable
- It saves us the effort of counting how long the list is
- It works, however long the list is

But in fact we want to do this so often there is a special type of loop for this:

```
marks=[34,45,72,19,23,78,42]
for m in marks:
    print(m)
```

This uses a different kind of loop, a *for loop*

```
for m in marks:
```

Inside the loop body, m takes on the value of each element in the list ‘marks’.

We can find the largest value in a list using the idea of the biggest so far, as in:

```
marks=[34,45,72,19,23,78,42]
biggestSoFar=0
for m in marks:
    if m>biggestSoFar:
        biggestSoFar=m
print(biggestSoFar)     # 78
```

But in fact Python already has a function to do this:

```
marks=[34,45,72,19,23,78,42]
print(max(marks))     # 78
```

Read the library reference for everything that can be done with lists:

https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

Maps – dictionaries

Another type of data structure is a map. This is a set of key-value pairs. We use it to look-up things. Given a key, we can look up the corresponding value that was paired with that key.

Maps are also called *dictionaries*. The idea is that we use them like we we use a dictionary, looking up a key word to find its value - the meaning of the word. But dictionaries work with any type, not just strings.

Python has a built-in map structure, which it calls a dict for dictionary.

For example, suppose we have a college management system, and we want a way to link students with their phone numbers. This would be a map:

```
phones=dict()
# add key value pairs
phones['John']='766 7896'
phones['Ahmed']='669 7006'
phones['Chi']='123 7796'
phones['Julie']='876 7096'
phones['Sean']='104 5432'
# get back some value
print(phones['Chi'])         # 123 7796
print(phones['John'])        # 766 7896
print('Jake' in phones)      # False
for name in phones.keys():   # all names
    print(name)
for number in phones.values():  # all numbers
    print(number)
```

In real code, we would not use student name as a key, because we could have two students with the same name. A key must be unique. We would use a student ID, probably an integer.

Files

We can store data in single variables, like

```
x = 3
```

or in structures such as lists

```
x = [5,6,2,8,9]
```

copyright © W W Milner 2021

but this data is held in program memory. When the program ends the memory is released and re-used by other programs, and when power is removed, all data in RAM is lost.

We often want data to be ‘kept’ which means it needs to be written to some non-volatile medium, such as a file held on disc.

Reading and writing files is actually done by the OS. Languages like Python do it by calling OS functions.

The OS uses things called file handles (Windows) or file descriptors (Unix) to keep track of what file is which. A file descriptor is set up by ‘opening’ a file, and released by ‘closing’ the file. All file use should be done in a sequence like

1. open the file
2. read and or write to the file
3. close the file

A *text file* is a file containing only a stream of text characters, stored as their character codes. The alternative is a *binary file* which contains data whish is not character strings, such as gif and jpeg and png image files.

Python code to write to a text file is like:

```
myFile = open("demo.txt", "w")
myFile.write("ABC 123")
myFile.close()
```

The open("demo.txt", "w") means to open the file named demo.txt, and the “w” is the mode, meaning to write to it.

Code to read back that data would be

```
myFile = open("demo.txt", "r")
str=myFile.read(3) # read just 3 characters
print(str) # ABC
myFile.close()
```

Exceptions

An exception is a situation or event not caused by program code, and which might result in unusual program operation, including termination.

Examples of exceptions are

- Running out of memory
- Trying to open a file which does not exist
- The user inputs invalid data
- A network connection is lost
- We try to write to a read-only file

One approach is to let an exception simply crash a program.

A better way is write code which deals with the exception somehow.
This means

- detecting when an exception occurs, then
- either taking action so it is dealt with - 'handling' the exception, or
- throwing or raising the exception. That means that the code which called the function in which the exception happened needs to handle it – passing the buck.

In Python, the try.. except construct is used to handle exceptions.

For example - getting user input, and dividing with it. If they enter 0 we would get a divide by zero exception. The code shown here catches the exception and outputs a message. This repeats until valid data is input:

```
fail=True                     # boolean flag - has input failed?
while fail:                   # repeat while its still failing
  try:
    x=int(input("Enter x: "))  # get input
    y=3/x
    fail=False                # everything OK now
    print(y)
  except ZeroDivisionError:
    print("Not zero, please")
```

copyright © W W Milner 2021

Structured programming

This is a *programming paradigm* – a set of ideas and techniques used to write programs. We can do structured programming in Python.

Before structured programming we had spaghetti code. Languages used a GOTO statement, which switched execution to a new place in code, not the next statement. This meant there were many sequences through a program, like a pile of noodles. With 10 or 20 lines of code this was not much of a problem, but with 100 or more, spaghetti code became a tangled web which was hard to write, hard to read, and likely to contain bugs.

In 1968 the computer scientist Edsgar Dijkstra published a letter headed 'GOTO considered harmful' describing these problems and the phrase entered programming folklore. Structured programming was the solution.

In structured programming we divide code into small units. These have different names in different languages, such as function, procedure, method or subroutine. Python calls them functions, which we use here. The idea is:

- A function is fairly small – no more than around 20 lines of code
- A function does just one thing
- Its name is what it does
- We 'call' a function – switch to it and let it run, as if it were a small program itself. At the end it does a 'return' - switches back to where it was called from.
- Program execution starts somewhere, and it will call functions
- A function can call another function
- Data can be passed into, and out of, a function.

If the problem is complex, we use a function to solve it, and avoid large functions by having the one function call others, so that each function can be fairly small.

This has many advantages. Each function is small so fairly simple. It can be tested by itself. A team of programmers can work together, each on one function. And a function might be re-used in other projects.

Python functions

For example:

```
def average(x,y):
    result=(x+y)/2
    return result

a=4
b=6
c=average(a,b)
print(c) # 5.0
```

This program is in two parts. We have the definition of the function:

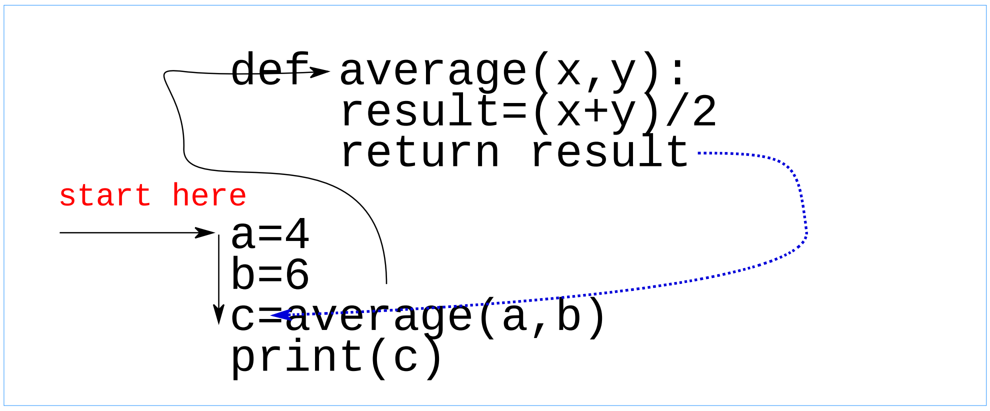
```
def average(x,y):
    result=(x+y)/2
    return result
```

This is just a definition. It says what the function average does. The function code is only executed when called.

The other part uses this function, as:

```
a=4
b=6
c=average(a,b)
print(c) # 5.0
```

The sequence of execution is:



The function definition comes first, but execution does not start there. It starts at the first code not in a function – the a=4.

At

```
c=average(a,b)
```

the function is called, and execution switches to the start of the definition. This continues until it hits the return statement, which makes the flow return to where it was called, and carry on from there.

Why not just say:

```
a=4
b=6
c=(a+b)/2
print(c)
```

We could, but if we need to find the average of lots of pairs of numbers, and many places in the program, it saves typing, and memory, to just define it once and call it whenever needed.

Its also clearer, explaining itself.

And we might be able to use that function in other projects.

Function parameters

These are sometimes called arguments. This is how data is passed *into* a function:

```
def average(x,y):
    result=(x+y)/2
    return result

a=4
b=6
c=average(a,b)
print(c)
```

The parameters are x and y, and a and b. x and y are sometimes called the *formal parameters* - the parameters in the function definition. a and b are sometimes called the *actual parameters* – the parameters used in the function call.

Check – the names differ. All that counts is the order. The value of a is passed to x, and b is passed to y.

Function returns

```
    return result
```

ends function execution, and passes back the value (in this case, the variable named ‘result’).

That returned value is used in this case by assigning it to c:

```
c=average(a,b)
```

Some functions do not return a value. They just do something – like deleting a file, or colouring a square red, or making a beep sound. In that case the function just ends:

```
  return
```

A function definition can have several returns. Function execution stops whenever a return is hit.

Some people say it is bad style to have more than one return in a function.

Scope

Scope means the region of source code in which a variable can be used.

For example:

```
def average(x,y):
    result=(x+y)/2
    return result

a=4
b=6
c=(a+b)/2
print(result)
```

The output is:

```
Traceback (most recent call last):
  File "/home/walter/Desktop/CompSci/PythonProgs/test.py", line 8, in <module>
    print(result)
NameError: name 'result' is not defined
Process terminated with an exit code of 1
```

We might think that

```
    result=(x+y)/2
```

is defining result – which it is. The problem is that result is *local* to the function average. Its scope is limited to the function, and is not available in the main code.

Why? Why is Python defined like that?

In real code we might have hundreds of functions, using variable names x,y, i, j, n, tax, result, color and so on. If variables were not local to a function, we would need to remember every variable used in every function, to avoid a clash and stopping one function wrecking another. For example:

```
   def average(x, y):
    result = (x + y) / 2
    return result


def bigger(x, y):
    if x > y:
        result = x
    else:
        result = y
    return result


c = average(4, 6)
d = bigger(9, 12)
print(d) # 12
```

Both functions use the variable named ‘result’. This is not a problem, because both are local, and do not affect each other.

If a variable is not local, it is *global* to the program, available everywhere. For example

```
def average(x, y):
    global total
    total=x+y
    result = (total) / 2
    return result

total=0
c = average(4, 6)
print(total) # 10
```

The variable total would have been local to average, but

```
    global total
```

changes this, and makes it global. So in the calling code, total is 10.

Pass by value

This means *copies* of parameters are passed to functions.

For example:

```
def average(x, y):
    total=x+y
    result = (total) / 2
    x=99
    return result

a=4
b=6
c = average(a, b)
print(a) # 4
```

Actual parameter a is passed to formal parameter x in the function, and inside the function, x is changed to 99. But that has not altered a. Why not?

Because a *copy* of a was passed to x. That copy was altered – but that did not change the original, so a stays at 4.

The alternative to pass by value is pass *by reference*, in which a pointer to the parameter is passed – then the function can follow the pointer and alter the value pointed at. But Python uses pass by value.

More on references later.

Built-in functions

The examples so far have been functions defined by the programmer.

The Python standard library has several useful functions already defined – like print().

They are documented at

https://docs.python.org/3/library/functions.html

Recursive functions

A function is recursive if it calls itself.

For example, the factorial function is the product of all the integers down to 1. This is written n!. So

3! = 3 X 2 X 1 = 6

4! = 4 X 3 X 2 X 1 = 24

How could we code factorial? One method would be

```
def factorial(n):
    if n == 1:
        answer = 1
    else:
        answer = n * factorial(n - 1)
    return answer


x = factorial(4)
print(x)  # 24
```

We can try to see how this works by adding print statements:

```
def factorial(n):
    print("In fact with n = ",n)
    if n == 1:
        answer = 1
    else:
        answer = n * factorial(n - 1)
    print('Returning ', answer)
    return answer


x = factorial(4)
print(x)  # 24
```

The output is

```
In fact with n =  4
In fact with n =  3
In fact with n =  2
In fact with n =  1
Returning  1
Returning  2
Returning  6
Returning  24
24
```

We start by calling fact(4). This calls fact(3), which calls 2, which calls 1. That returns 1 – to fact(2), which returns 2 to fact(3), which returns 6 to fact(4), which finally returns 24.

Suppose we change this to

```
def factorial(n):
    answer = n * factorial(n - 1)
    return answer


x = factorial(4)
print(x)
```

Output is:

```
Traceback (most recent call last):
  File "/home/walter/Desktop/CompSci/PythonProgs/test.py", line 6, in <module>
    x = factorial(4)
  File "/home/walter/Desktop/CompSci/PythonProgs/test.py", line 2, in
factorial
    answer = n * factorial(n - 1)
```

```
..
RecursionError: maximum recursion depth exceeded
Process terminated with an exit code of 1
```

Now fact(4) calls 3, which calls .. 1, 0, -1, -2 and so on. This is *infinite recursion*, which would never end, but in fact the system stores return information in memory, and eventually runs out of space.

Actual recursive code always has a conditional so that eventually the recursion ends.

Any recursive code can be converted to iteration – a loop:

```
def factorial(n):
    result=1
    for i in range(1,n+1): # so 1,2,3..n inclusive
        result*=i
    return result


x = factorial(4)
print(x)
```

Object-oriented programming

This (OOP) is a paradigm, different from structured programming.

Objects

An object is a piece of information, in memory, which is a bundle of data members and pieces of code.

The data members are called *fields*. The pieces of code are called *methods*.

Objects have special methods, called *constructors*, which are used to make new objects.

Python calls all object members *attributes*. Data members are called *properties*, and method are called *callable attributes*.

Classes

A class is a *type of object*.

An object is an *instance of a class*. We have lots of objects about the same type of thing. They have the same class.

Not all OOP languages have classes. Python does.

Python has several useful classes built-in to the standard library, and we use these all the time. But we can also define and use our own classes.

A Student class

Suppose we want a college management system, and as part of that, we want student records.

We have a lot of students, so we would have a class Student, which would be the type of every student. Each student would have a name, a unique ID, and a course (and many other attributes, but this is a simplified example).

We explain this code below:

```
 # define the class
class Student:
```

copyright © W W Milner 2021

```
    def __init__(self, n, i, c):
        self.name=n
        self.id=i
        self.course=c

    def setCourse(self, c):
        self.course=c

# code to use the class
stud1=Student('Ahmed', 253, 'Chemistry')
print(stud1.course) # Chemistry
stud1.setCourse('Physics')  # change course
stud2=Student('James', 617, 'Engineering') # another student
```

Methods

A method is a piece of code, like a function, which is part of an object.

Our Student class has a method named setCourse:

```
    def setCourse(self, c):
        self.course=c
```

Student objects have a field named ‘course’. The ‘self’ parameter refers to the object executing the method. So when we invoke the method, with

```
stud1.setCourse('Physics')
```

then self refers to object stud1. And self.course means the course field of that object.

c is a simple parameter as in a normal function call. So in

```
stud1.setCourse('Physics')
```

the actual parameter ‘Physics’ is copied to the formal parameter c.

Constructors

A constructor is a special method, used when a new object is being created.

In Python, constructors have the special name __init__. That is 2 underscore characters, _, init, and another 2 underscores _. The weird name __init__ is intended to avoid clashes with method names we might choose.

Our constructor is:

```
    def __init__(self, n, i, c):
        self.name=n
        self.id=i
        self.course=c
```

and is used by

```
stud1=Student('Ahmed', 253, 'Chemistry')
```

As in a normal method, 'self' refers to the object – in this case, stud1.

The constructor just assigns values to the object fields – name, id and course.

Special method names

Python classes have a set of special method names, informally called 'dunder methods'. __init__ is an example. They start and end with 2 underscores. The list is documented at

https://docs.python.org/3/reference/datamodel.html#specialnames

A common use is __str__. If we print out an object, this happens:

```
class Student:
    def __init__(self, n, i, c):
        self.name=n
        ..

# code to use the class
stud1=Student('Ahmed', 253, 'Chemistry')
print(stud1)
```

```
<__main__.Student object at 0x7f36bf308908>
```

which is not very informative.

In fact print calls str(..), a built-in function which converts its parameter to a string. In turn str calls the __str__ dunder method. So we can define a __str__ method like:

```
 # define the class
class Student:
    ..
    def __str__(self):
        result='Student: '
        result+=self.name
        result+=' ID:'+str(self.id) # convert int to string
        return result

# code to use the class
stud1=Student('Ahmed', 253, 'Chemistry')
```

```
print(stud1)
```

we get

```
Student: Ahmed ID:253
```

Per class fields

Fields can be per object. That is, each object has a value for a field.
Each student has their own name.

But it is sometimes useful to have fields which are about the class, not about each object. These are sometimes called *static* fields.

For example, we can say

```
stud1=Student('Ahmed', 253, 'Chemistry')
stud2=Student('Jim', 253, 'Biology')
```

In other words we can have 2 students with the same ID which should be impossible. We can fix this by having the class store the last ID used, and increment that automatically every time an object is created:

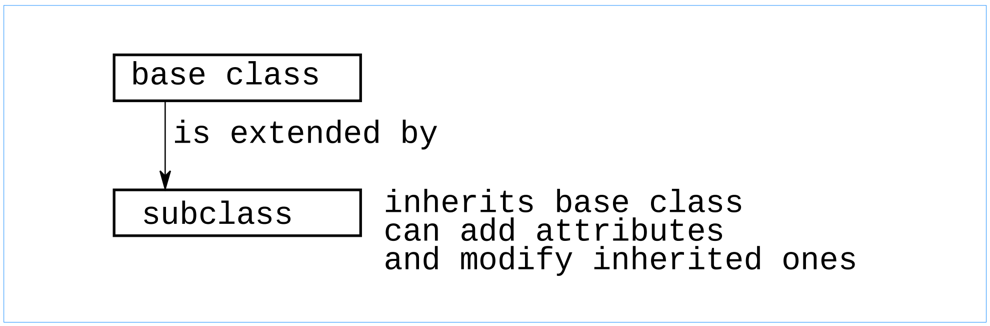
```
class Student:
    lastIDUsed=0
    def __init__(self, n,c):
        self.name=n
        self.course=c
        Student.lastIDUsed+=1
        self.id=Student.lastIDUsed
    def setcourse(self, c):
        self.course=c
    def __str__(self):
        result='Student: '
        result+=self.name
        result+=' ID:'+str(self.id) # convert int to string
        return result


stud1=Student('Ahmed','Chemistry')
stud2=Student('Jim','Biology')
print(stud1)
print(stud2)
```

```
Student: Ahmed ID:1
Student: Jim ID:2
```

Inheritance

This is about relationships between classes. The idea is to have a base class, related to a subclass. The subclass inherits the attributes of the base class, like a child inheriting from a parent. The subclass can add extra attributes, or midfy the inherited ones.



The purpose of this is to re-use code. When we define a new class, we may be able to extend an existing one to re-use its code.

This is done by syntax like

```
class Sub(Base)
```

Then Sub inherits the members of Base, and they can be altered if needed.

For example, we might have a college management system. After analysing the requirements, we might decide

- We need to handle data on people, using a class Person. Each person has a name.
- One type of Person is a Student. A Student also has a student ID.
- A different kind of Person would be Staff. A staff member would have a tax code.

```
class Person:
  def __init__(self, name):
    self.name = name

  def setName(self, name):
    self.name = name

  def getName(self):
    return self.name

  def __str__(self):
```

copyright © W W Milner 2021

```
    return "Person: name=" + self.name


class Student(Person):
  lastIDUsed = 100

  def __init__(self, name):
    self.name = name
    self.id = Student.lastIDUsed
    Student.lastIDUsed += 1

  def getID(self):
    return self.id

  def getLast():
    return Student.lastIDUsed

  def __str__(self):  # over-ride method
    return "Student: name=" + self.name


stud1 = Student("John")
stud2 = Student("Jane")
stud3 = Student("June")
print(stud2.getName())  # use an inherited method
print(stud3)            # Student: name=June
```

The class Student inherits the method getName, and has an additional attribute, id.

The class Staff would have an additional attribute of taxCode.

Class hierarchies

In practice we often have not just one base class and one subclass. The subclass is often in turn the base of another subclass. This is like multiple generations in a family tree, of grandfather, father and son.

Such a thing is called a class hierarchy.

Over-riding methods

When we code a method in a subclass, with the same name as a method in a base class, this is called over-riding a method.

In this example we have the base class method:

```
  def __str__(self):
    return "Person: name=" + self.name
```

and this is over-ridden by the subclass:

copyright © W W Milner 2021

```
  def __str__(self):
    return "Student: name=" + self.name
```

When we invoke the method (indirectly through print)

```
print(stud3)
```

it uses the subclass method.

This is useful, because it means we can say

```
print(anyone)
```

and that will use the version of __str__ in whichever class anyone belongs to – we do not need to know which that is.

This is called polymorphism. We define different versions of a method with the same name in different subclasses. We can then invoke the method by name, and the appropriate version will be used.

This avoids code like

If the type of an object is X, do this.

If the type is Y, do something else

If the type is Z, do this..

Such code is always wrong.

Abstract base classes

An abstract base class is a class towards the top of a class hierarchy which is very general, and is not intended to be instantiated.

For example in our college management system we have a base class Person and subclasses Staff and Student. Any college member must be either a Staff member or a Student – they cannot just be a Person. So Person is an abstract base class.

Not So Basic Ideas

All data are objects

For example

```
x = 5
print(type(x)) # <class 'int'>
```

Every object has a set of attributes. Some are callable, the rest are properties

All named code is data

Named code includes class methods, user-defined functions, and built-in fuctions. Since they are data, they are also objects:

```
print(type(print))  # <class 'builtin_function_or_method'>
```

Useful built-in functions

Some functions relevant to objects are

```
print(type(print))  # <class 'builtin_function_or_method'>
print(callable(print))  # True
print(dir(print))
# list all attributes of print
# get ['__call__', '__class__', '__delattr__', '__dir__', '__doc__',
# '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
# '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
# '__module__', '__name__', '__ne__', '__new__', '__qualname__',
# '__reduce__', '__reduce_ex__', '__repr__', '__self__', '__setattr__',
# '__sizeof__', '__str__', '__subclasshook__', '__text_signature__']
print(id(print))  # In CPython, the address of print
```

The object class

There is a class named object, and all other classes are sub-classes of it. As a result, all objects inherit these attributes:

```
print(dir(object))
# ['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
# '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
# '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__',
# '__new__', '__reduce__', '__reduce_ex__', '__repr__',
# '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

Special name attributes

All the attributes of object have 'special names', starting and ending with double underscore __.

Some of these are callable:

```
print(callable(print.__str__)) # True
print(callable(print.__doc__)) # False
```

and when we call a callable:

```
print(print.__str__())  # <built-in function print>
```

so __str__ is a method which returns a string version of the object.

But __doc__ is not callable:

```
print(print.__doc__)
```

so __doc__ is a non-callable field, documenting the object. The output is:

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

As data, print.__doc__ is an object:

```
print(type(print.__doc__)) # <class 'str'>
```

But usually dunder methods are usually called indirectly. So the built-in function str calls the dunder method __str__

```
print(str(print))  # <built-in function print>
# or more usually:
x = 4
y = 5
print(str(x) + str(y)) # 45
```

Operator overloading

An operator is like + and -.

Overloading means re-using the same name to do different things with different types.

For example, + is already overloading. It adds ints – 4+5=9, but concatenates strings 'good'+'boy' = 'goodboy'.

Programmatic operator overloading means writing code which will overload an operator as we wish. C++ can do this, and so can Python, using special name methods.

For example we can define a vector class, and override the __add__ method. The interpreter calls __add__ when it sees + in an expression:

```
class ThreeVector:
  def __init__(self, x,y,z):
    self.x=x
    self.y=y
    self.z=z

  def __add__(self, other):
    result=ThreeVector(self.x+other.x, self.y+other.y, self.z+other.z)
    return result

  def display(self):
    print(self.x,',',self.y,',',self.z)

v1=ThreeVector(3,2,4)
v2=ThreeVector(2,3,5)
v3=v1+v2
v3.display() # 5,5,9
```

Encapsulation and access control

Encapsulation is a key idea in OOP. It means data and code is packaged into objects, and objects are sealed and access to what is inside each object is controlled. There is no global data, only data with objects, accessed by the code in the object.

Here is a problem. Suppose we have a graphics application, and we define a Circle class:

```
class Circle:
    def __init__(self, r):
        self.radius = r
        self.x = 0    # co-ordinates of the centre
        self.y = 0    # default values


myCircle = Circle(50)
myCircle.radius = -4
```

We made a circle with radius 50. Then we change it to radius 4. Why did we do that? Maybe it was a mistake. Maybe we got confused. Maybe we meant myCircle.x=4.

The point is that a negative radius makes no sense. It should be impossible. It puts the object into an *invalid state*. We would like a language feature which would make this impossible.

The attributes of the circle object can be accessed from anywhere. They are in effect global data – against the whole point of OOP.

The usual solution is

- Set the object fields to be *private* - that is, only accessible from within the object
- Have *getter access methods* to provide public *read* access to those field. Like

```
def getRadius(self):
  return self.radius
```

- Have *setter access methods* which allow public *write* access to set field values, but only if they are valid. Like

```
def setRadius(self, r):
  if r>0:
    self.radius=r
```

This makes sure the object state is always valid.

Unfortunately Python does not have such a language feature – fields cannot be set to be private.

One solution is the convention that fields starting with a single underscore should be treated as private. If code directly access such a field, it might go wrong, especially in future versions.

So, for example:

```
class Circle:
    def __init__(self, r):
        self._radius = r
        self._x = 0    # co-ordinates of the centre
        self._y = 0    # default values

    def getRadius(self):
        return self._radius

    def setRadius(self, r):
        if r > 0:
            self._radius = r


myCircle = Circle(50)
myCircle.setRadius(4)       # call setter method
myCircle.setRadius(-9)      # -9 is ignored, as invalid
print(myCircle.getRadius()) # 4 - call the getter method
```

Another solution is to use the @property decorator.

In the following code, we have named the radius field _obscure, with the idea that no-one is likely to access that accidentally. But we use @property to establish radius as a property, and code the setter to validate it:

```
class Circle:
    def __init__(self, r):
        self._obscure = r
        self._x = 0    # co-ordinates of the centre
        self._y = 0    # default values

    @property
    def radius(self):
            return self._obscure

    @radius.setter
    def radius(self, value):
        if value > 0:
            self._obscure = value


myCircle = Circle(50)
myCircle.radius = 4         # call setter method
myCircle.radius = -9        # is ignored
print(myCircle.radius)      # 4 - call the getter method
```

Bytecode

How does Python work?

Python is a grammar. Whatever executes Python code according to that grammar can do it however it likes.

But if we use the CPython interpreter, that will

1. Compile Python source code to bytecode, then
2. Execute that bytecode on a virtual machine

Code objects like functions have a special name attribute __code__, which in turn is an object with several fields:

```
def myFunc():
    x = 6
    y = 27
    z = x + y
    return z
```

copyright © W W Milner 2021

```
theCode = myFunc.__code__
print(theCode.co_consts) # (None, 6, 27)
print(theCode.co_varnames) # ('x', 'y', 'z')
print(theCode.co_code)
# b'd\x01}\x00d\x02}\x01|\x00|\x01\x17\x00}\x02|\x02S\x00'
```

The co_consts attribute is a tuple of the constants referenced in the code. It starts with None because functions default to returning None, so is often needed, so the compiler puts it there.

co-varnames is a tuple of the variables referenced.

co_code is the bytecode. This is pretty unreadable. But Python has a disassembler, which can convert the bytecode into mnemonics, which allow us to see how it works:

```
import dis


def myFunc():
    x = 6
    y = 27
    z = x + y
    return z

dis.dis(myFunc)
```

Output is

```
  5           0 LOAD_CONST               1 (6)
              2 STORE_FAST               0 (x)

  6           4 LOAD_CONST               2 (27)
              6 STORE_FAST               1 (y)

  7           8 LOAD_FAST                0 (x)
             10 LOAD_FAST                1 (y)
             12 BINARY_ADD
             14 STORE_FAST               2 (z)

  8          16 LOAD_FAST                2 (z)
             18 RETURN_VALUE
```

The first line is

```
  5           0 LOAD_CONST               1 (6)
```

5 is the line number in the original Python source code

0 is the offset into bytecode, in bytes. These go up in steps of 2, because each each operation is an opcode (one byte, hence the name bytecode) and a one byte operand

1 is the operand – here the offset of the constant in the constants tuple

(6) conveniently tells us what that is.

The Python virtual machine is a stack machine, with instructions making heavy use of an evaluation stack. The code with explanations is:

```
LOAD_CONST 1 (6)  get constant at index 1 and push onto stack
STORE_FAST 0 (x)  pop stack into local variable index 0 – IOW x=6

LOAD_CONST 2 (27) push 27 on stack
STORE_FAST 1 (y)  pop to index 1 – so y=27

LOAD_FAST  0 (x)  push index 0 onto stack – so push x
LOAD_FAST  1 (y)  push y
BINARY_ADD        pop 2 off stack, add push result back
STORE_FAST 2 (z)  pop stack to z – so z=x+y

LOAD_FAST 2 (z)   push z on stack
RETURN_VALUE      pop stack and return it IOW return z
```

A full list of bytecode instructions is at

https://docs.python.org/3.9/library/dis.html#python-bytecode-instructions

copyright © W W Milner 2021

Glossary

bit	A binary digit – a 0 or a 1
byte	A group of 8 bits
compiler	Software which inputs code in one language (source code) and outputs an equivalent program in another language (object code). Often source code is a high level language (such as C) and object code is a low level language.
disassembler	Software which converts native code into mnemonics
element	Value in a data structure such as a list
integer	A whole number. This might be unsigned, like 348, or a signed integer like +48 or -3244
interpreter	Software which inputs a program (often called a script) and executes it instruction by instruction.
iteration	Looping. Repeating code
native code	A processor was a set of instructions which it can recognise and execute directly. Such instructions are every simple – like add integers or move bytes. Each instruction consists of an op-code – an operation code – and an operand. Both are binary patterns. Native code is made of such instructions.
OS	Operating system, a set of software items needed to make computer hardware usable. Examples are Microsoft Windows, distributions of Linux like Ubuntu and Debian, Android, MacOS and iOS.
volatile	A storage medium which loses its data when switched off, such as RAM. Non-volatile media keep data when switched off, such as disc files.