Python Notes

Table of Contents

Background and setup

This is a set of brief notes about Python.

It does not explain underlying ideas which are general to all languages. For example it does not explain what an interpreter is or what OOP is. Read the theory papers for that.

These notes set out how those ideas are used in Python.

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.

If you are setting up your own device to write and run Python, you need

- A Python interpreter, which executes Python code, and
- An editor to write Python source code.

You may already have the interpreter. You can find out by starting a terminal and typing:

```
python3 -V
```

to show the version number. If it says 'No such program' or similar you need to download and install it.

Go to

https://www.python.org/downloads/

and install the correct version for your operating system.

You can write a Python program using any text editor (not a word processor), then tell the Python interpreter to run it.

For example in notepad or geany or another text editor, type in

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

and save it in a suitable folder with a name like myprog1.py.

Then at the command line in that folder, invoke the interpreter and make it execute that script by saying:

```
python3 myprog1.py
5
```

and get output 5, as expected.

But in Python, text indentation is syntactically significant. That means the layout of your code is very important. For example

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

```
print("Finished")
```

the second line is indented. This is because it is the body of a loop, which starts at line 1. Line 3 is not indented, because it is after the loop.

Lines can be indented using SPACE characters (hit the Space Bar), or by TAB characters (hit the Tab key). An editor might convert the TAB to 2 or 4 or 8 spaces (depending on how it is configured). Or it might keep it as an actual tab character. It might display white space (tabs and spaces) or it might not.

But the tabs and spaces should be consistent. Use one or the other. Do not mix them.

This might be confusing. One fix is to use an IDE configured for Python. One choice is Wing:

https://wingware.com/

The personal version is free. It will do indentation for you, and you can execute the script in the IDE.

Language reference and tutorials

The reference to the language is

https://docs.python.org/3/reference/index.html

The reference to the standard library is

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

These are references, so they list everyting in full detail. But they explain nothing. That is not the purpose.

A tutorial, which does explain things, is here:

https://docs.python.org/3/tutorial/index.html

Beware of Python blogs written by students and journalists. They are often mis-leading. Use python.org.

This text is about Python 3.

> *Things to do are formatted like this. Try them out to check your understanding.*

Links to resources

The web is full of Python blogs, many of them written by beginners, full of errors and inaccuracies. Use the material either from the official source, python.org, or from a top-rank university.

These links might change. Please report broken links to w.w.milner@gmail.com

Beginner's tutorial including setup
python.org's beginners page - has more links
The language reference – not for beginners
The Python Foundation official website
Python from MIT free courseware
Khan Academy youtube Python videos
A wikibook on Python
Python ideas
Module of the week
Wing IDE

Using built-in classes

Classes objects and methods

Python is a pure OOP language. That means that all values are objects. An *object* has some data, and also some methods. A method is some code which can do something. An object is a bundle of code and data.

Objects belong to some *class*. A class is a type of object. In Python the type() function can be used to output what type an object belongs to.

For example

```
x=3
print(type(x)) # this outputs <class 'int'>
y=6
print(type(y)) # this also outputs <class 'int'>
z="Hello"
print(type(z)) # this outputs <class 'str'> : str=string
```

Copy this program and run it to check the output. Try and find other types.

Comments

Comments are text in source code which is ignored by the interpreter or compiler.

In Python, comments start with #. See the example above.

Methods

We tell an object to execute one of its methods using the 'dot syntax', like

```
object.method()
```

The string class has several methods. For example

```
str="hello"
str2=str.capitalize()
print(str2) # Hello
```

The string class is *immutable*. That means once we make a string, we cannot change it. What capitalize does it to *make a new string*, which is like the one executing the method, but with the first letter changed to upper case, and returns this new string. Then

```
str2=str.capitalize()
```

assigns this *new string* to str2.

We can say

```
str=str.capitalize()
```

but that means we are assigned str to this new string - not changing the old string.

Look at https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str to find the methods of class string. Write a short Python program to try them out.

Number classes

Python has 3 built-in number types. These are integer, floating point and complex. These have a few methods - for example:

```
r=1.25 # float
p=r.as_integer_ratio()
print(p) # (5, 4) : 1.25 = 5/4
i=17 # integer
bits=i.bit_length()
print(bits) # 5 : 17=10001 in base 2
```

The usual arithmetic operators work with number objects:

```
x=2*3+4
print(x) # 10
```

Internal representations of numbers

How are numbers represented in binary in Python?

Python code is normally executed by an interpreter. Those interpreters are usually written in C. Floating-point types are represented how they are in that version of C - which might vary. If we need to know, sys.float_info can tell us:

```
import sys # need this module

print(sys.float_info)
# sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
# max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
# min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, # radix=2,
rounds=1)
```

This is using IEEE 754 floating point format, with a mantissa of 53 bits and exponent with 15 bits, a maximum value of $1.7976931348623157 \times 10^{308}$ and so on.

Integers are held in a different way. They are held as a sequence of digits:

```
print(sys.int_info)
# sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

but these are not decimal (base 10) digits. Each digit is held in 30 bits, so is base 2^{30}. Each digit is in 4 bytes - the size of a typical C int. The number of digits is indefinite - so there is no maximum to the size of an integer in Python.

Representation of characters

Python does not have a character type. It just has a string class, and a character is a string with length 1.

Within a string object, each character is represented by its Unicode code-point. ord() tells you the code point of a character, and chr gives you a character from its code point:

```
myChar="A"
print(ord(myChar)) # 65
myChar="θ"
print(ord(myChar)) # 952
myChar=chr(97)
print(myChar) # a
```

Built-in functions

As well as object methods, Python has a set of functions which are built-in to the language - that is, they do not require any library code.

Examples are print, ord and chr.

The full list is at https://docs.python.org/3/library/functions.html

Boolean class

For example:

```
x=True
print(type(x)) # <class 'bool'>
```

Boolean class objects can only have values True and False (check the Capital letters)

Some expressions have values which are boolean type:

```
print( 5>3 ) # True
```

Lists tuples and ranges

These 3 classes are 'sequence types' as follows:

- A list is a line of values which can grow and shrink and be split
- A tuple is fixed when it is created and cannot change - it is read-only
- A range is a line of values in equal steps, stored compactly

For example

```
myList=[91,22,53,64]
```

```
print(myList[0])    # 91 - first element is index 0
print(myList[1:3])  # 22,53 - 3 is teh first element to ignore
myList.append(99)
print(myList) # [91, 22, 53, 64, 99]
print(53 in myList) # True - 53 is in myList
myList.remove(22)
print(myList) # [91, 53, 64, 99]
```

Check the *[square brackets] around a list*

A tuple is:

```
myTuple=(4,5,6)
print(myTuple[1]) # 5
# myTuple[1]=7 "TypeError: 'tuple' object does not support item assignment"
# myTuple.remove(5) "AttributeError: 'tuple' object has no attribute 'remove'"
print(len(myTuple)) # 3
```

Check the *(round brackets) for a tuple.*

A range:

```
myRange=range(1,9,2)
# from 1 in steps of 2 up to 9 exclusive
# so 1 3 5 7
print(myRange[2]) # 5
myRange=range(-3,3) # step defaults to 1
print(myRange[2]) # -1
```

A range only stores the limits, so range(1,1000) takes no more space than range(1,5)

Range is most often used in a for loop, described later.

Lists and tuples are similar to *arrays* in some languages, in that we can get an element directly using an index. But lists can change in size, and tuples are read-only.

Declarations

Python variables do not need to be declared.

But if you use a variable before it has an assigned value, the interpreter will produce an error message:

```
x=23
z=x+y
print(z) # builtins.NameError: name 'y' is not defined
```

Types and references

In Python, values (objects) have type. Variables do not have type. The type of the value of a variable can change, if its value changes:

```
x=5
```

```
print(type(x))    # <class 'int'>
x="Hello"
print(type(x))    # <class 'str'>
print(type(6.7))  # <class 'float'>
```

A Python assignment, like

```
x=5
```

makes it look like x is 5. Its not.

x is a *reference* (pointer) to an object - an instance of the class 'int'. That object contains a value 5.

A reference is a general trm for a value which somehow allows a system to find where something is. In this case, the reference is to find an object.

Python has a function named id, which returns a unique and constant value for the object. For the CPython implementation, this is the address in RAM of the object:

```
1   x=5
2   print(hex(id(x))) # 0xa68de0 - the address of the x object, in hex
3   y=5
4   print(hex(id(y))) # 0xa68de0
5   print(x==y)       # true
6   print(x is y)     # true
7   x=21
8   print(hex(id(x))) # 0xa68fe0
9   print(y)          # 5
```

At line 1, it creates a 5 object, and sets x to point to it.

At line 2 we output the id of x - its address. The hex formats that in base 16.

Line 3 assigns 5 to y

Line 4 shows the address of the y object is the same as x - they both point to the same object

Line 5 shows the value of x is the same as y (both 5)

Line 6 confirms the x object *is* the y object - they both point to the same object.

At line 7 we assign x to a different object

Line 8 shows its different address

Line 9 shows this has not affected the y object.

The built-in types are *immutable*:

```
x=5.1
print(hex(id(x))) # 0x7fd733993960
x=x+0.3
print(hex(id(x))) # 0x7fd733993978
```

so x=x+0.3 does not change the x object. It makes a new object, and makes x point to it.

Type-casting

Type casting means changing the type of an object. That is impossible in Python.

But some of the built-in functions take one value, and create a new object of a different type:

```
x=int(3.1) # 3
x=float(4) # 4.0
x=float("4.5") # 4.5
# x=float("4.5.6") "builtins.ValueError: could not convert string to float: '4.5.6'"
x=str(3.1) # "3.1"
```

Conditional statements

In Python these are like this:

```
x=10
if x>8:
  print("Its less than 8")
else:
  print("its 8 or more")
print("This is after the if")
```

Check:

- No brackets around x>8 (unlike C, JavaScript, Java and so on)
- A colon : at the end of the if header
- The indentation of the block
- The colon : after the else
- The indentation of the else block
- No endif. The end is marked by the end of the indentation.

This is why indentation of source code is very important in Python.

Another example:

```
x=10
y=2
if x!=8 or (y<4 and x>5):
  print("The if is true")
```

```
  x=0
else:
  print("The if is false")
  x=1
print(x)  # 0
```

Check the use of or and and. The if is true because x is not equal to 8.

Try out the above program.
What are de Morgan's Laws?
Re-write this without using not:

```
x=9
y=3
if not(x==10 or y==2):
  print("The if is true")
```

Loops - while

There are 2 kinds of loops in Python - while and for. For example

```
# add up numbers 1 to 100
total=0
counter=1
while counter!=101:
  total=total+counter
  counter=counter+1
print(total)  # 5050
```

Check

- The colon : at the end of the loop header
- The indentation of the loop body

This would more usually be written:

```
# add up numbers 1 to 100
total=0
counter=1
while counter!=101:
  total+=counter  # same as total=total+counter
  counter+=1      # same as counter=counter+1
print(total)  # 5050
```

Write a program to add up the odd numbers from 1 to 99 inclusive

The for loop is used with an *iterable type*, such as lists tuples strings and ranges. Iterable means we can get each element at a time in a sequence, using the keyword 'in':

```
myList=[3,-9,4]
for x in myList:
  print(x) # 3 -9 4 on separate lines
myTuple=(4,3,2)
for x in myTuple:
  print(x)  # 4 3 2
for x in "aceg":
  print(x) # a c e g
```

But for is most often used with a range sequence:

```
for n in range(1,5):
  print(n) # 1 2 3 4
```

Use a for loop to add up the integers 1 to 10000

2D arrays

There are no arrays in Python, but we can use lists, and make a 2D list as a list of lists.

We can also use a nested loop to iterate through all elements:

```
 # 2D arrays in Python
row0 = [4, 3]
row1 = [1, 2]
matrix = [row0, row1]
print(matrix[1][1]) # 2
for row in matrix:
  for element in row:
    print(element) # 4 3 1 2
```

Defining new functions

As well as using the built-in functions, we can define and use our own:

```
# functions
def average(a,b): # function definition starts here
  result=(a+b)/2
  return result

x=3 # execution starts here
y=4
z=average(x,y)
```

```
print(z) # 3.5
```

Alter this so it finds the average of 3 values

We need to think about *lifetime*

```
def average(a,b): # function definition starts here
  result=(a+b)/2
  return result


x=3 # execution starts here
y=4
z=average(x,y)
print(z) # 3.5
print(result) # no - get "builtins.NameError: name 'result' is not defined"
```

The variable 'result' is *local* to the function. That means its lifetime begins when the function starts to execute, and ends when the function ends. So we cannot refer to it after the function has ended - it no longer exists.

Also being local, it has no connection with a global value with the same name:

```
def average(a,b): # function definition starts here
  result=(a+b)/2
  return result

result=99
x=3 # execution starts here
y=4
z=average(x,y)
print(z) # 3.5
print(result) # still 99
```

But Python is unusual, in that the keyword 'global' means that a variable used in a function is not local:

```
def average(a,b): # function definition starts here
  global result
  result=(a+b)/2
  return result

result=99
x=3 # execution starts here
y=4
z=average(x,y)
print(z) # 3.5
print(result) # now 3.5
```

We can pass any type as a parameter:

```
def sum(aList):
  total=0
  for num in aList:
    total+=num
  return total

myList=[1,2,3]
print(sum(myList)) # 6
```

In Python, *parameters are passed by value*. That is, a *copy* of the parameter is passed to the function.

But in Python, variables are pointers to objects, not objects themselves.

So a *copy of the pointer* is passed, not a *copy of the object*. This is for speed - the object might be very large, and copying it could be slow and use a lot of memory. This is the same as Java and JavaScript.

For example

```
def myFunction(x):
  print(id(x)) # 10915520
  return

y=12
myFunction(y)
print(id(y)) # 10915520
```

Here y is a pointer to a 12 object. The object is stored at address 10915520. A copy of that pointer is passed to the function. So in the function, x also points to address 10915520 - the same 12 object.

That means that if the object is mutable, the function can follow the pointer and alter it:

```
def sum(aList):
  total=0
  for num in aList:
    total+=num
  aList[0]=99
  return total

myList=[1,2,3]
sum(myList)
print(myList[0]) # 99
```

Give examples of mutable and immutable types

Recursive functions

These work as in other languages. For example

```
def factorial(n):
  print("Entering factorial with n = ",n)
  if n==1:
    return 1
  result=n*factorial(n-1)
  print("Leaving factorial with n = ",n)
  return result

print(factorial(4))
```

Predict the output of the above program
Run it to check
Convert it to an iterative version

Writing our own classes

Suppose we want to write a program about college students. We need to define a class which will model a student:

```
class Student:
  def __init__(self, name, id): # constructor
    self.name = name
    self.id = id


stud1 = Student("John", 328) # invoke constructor
stud2 = Student("Jane", 241)
print(stud1.id) # 328
print(stud2.name) # Jane
```

The lines in *italics* define the class. The rest uses the Student class.

It is usual to name classes starting UpperCase.

Constructors

A constructor is a special method used to run code when making an objcet - a new class instance. The constructor has the special name __init__ (two underscores _ and _ before and after).

The first parameter to __init__ is a reference to the object being constructed. This is usually named self (it does not have to be). In other languages it is called 'this'.

A class is a type of object. An object is an instance of a class.

Most classes have attributes. This example has two attributes - name and id.

```
    self.name = name
```

means to assign name to the name attribute of the object being constructed.

We could have said

```
  def __init__(self, x, y): # constructor
    self.name = x
    self.id = y
```

but it is more common, and more readable, to choose parameter names to suggest what they represent.

When we say

```
stud2 = Student("Jane", 241)
```

this constructs a new object, with class Student, with name Jane and id 241. The variable stud2 points to this new object.

> *Define a class named Course. This should have attributes subject and roomNumber, so you can say things like*
> *course1=Course("Physics",28)*
> *course2=Course("Chemistry",32)*

Methods

A method is defined as a function within the class definition. The first parameter in the definition is a reference to the object invoking the method:

```
class Student:
  def __init__(self, name, id):
    self.name = name
    self.id = id
  def display(self): # define method
    print("Student: Name= ",self.name," ID = ", self.id)

stud1 = Student("John", 328)
stud2 = Student("Jane", 241)
```

```
stud2.display() # invoke method
```

A method can declare local variables, contain any kind of statement, and return a value:

```
class Student:
  def __init__(self, name, id, grade1, grade2):
    self.name = name
    self.id = id
    self.grade1=grade1
    self.grade2=grade2
  def display(self):
    print("Student: Name= ",self.name," ID = ", self.id)
  def averageGrade(self):
    result=(self.grade1+self.grade2)/2
    return result

stud1 = Student("John", 328, 17,23)
stud2 = Student("Jane", 241, 20, 40)
print( stud2.averageGrade()) # 30.0
```

Try out this code.
Add a method getID() which returns the student's ID.
Test it

Special method names

All Python classes have a set of special methods, with names enclosed with __ double underscores__

These are called by other 'normal' functions. For example, print calls __str__ to get a string version of the object to display it.

See

https://docs.python.org/3/reference/datamodel.html#special-method-names

and operator over-loading below.

Encapsulation

Python has no way of preventing direct access to class members, so cannot enforce encapsulation.

But there is a convention that members with names starting with underscores are intended to be treated as private members, not to be accessed directly.

In that case, getter and setter methods might be provided:

```
class Student:
  def __init__(self, name, id):
    self.__name__ = name
    self.__id__ = id

  def setName(self,name):
    self.__name__=name
  def getName(self):
    return self.__name__

stud1 = Student("John", 328)
stud2 = Student("Jane", 241)
print( stud2.getName()) # Jane
```

Per class variables

In OOP ideas, instance variables are per object - each object can have a different value. There are also per class variables, where the class as a whole shares a single value.

The following example shows this idea used to make sure no 2 students get the same ID:

```
class Student:
  lastIDUsed=100

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

  def setName(self,name):
    self.__name__=name
  def getName(self):
    return self.__name__
  def getID(self):
    return self.__id__

stud1 = Student("John")
stud2 = Student("Jane")
stud3 = Student("June")

print( stud3.getID()) # 102
```

The class has a class-level field named lastIDUsed. In the constructor, this is used as the ID, and it is then incremented ready for the next student. The constructor does not have an ID parameter, because it is not needed.

Check that the class-level field is accessed as <Classname>.<fieldName>, as for example Student.lastIDUsed.

> *Use the Courses class defined earlier. Include an auto-assigned ID field using this technique.*

Per class methods

These are also called static methods, as used in Java:

```
class Student:
  lastIDUsed=100

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

  def setName(self,name):
    self.__name__=name
  def getName(self):
    return self.__name__
  def getID(self):
    return self.__id__
  def getLast(): # no self
    return Student.lastIDUsed

stud1 = Student("John")
stud2 = Student("Jane")
stud3 = Student("June")
print(Student.getLast()) # 103
```

These are invoked as <class><method>, like Student.getLast()

Overloading constructors and methods

There are several ways to achieve constructor and method over-loading. One approach is to use Python's ability to have default parameter values, and in particular, None. For example

```
class Student:
  lastIDUsed=100

  def __init__(self, name=None):
    if name is None:
      self.__name__="Name unknown"
    else:
      self.__name__ = name
    self.__id__ = Student.lastIDUsed
    Student.lastIDUsed+=1
```

```
  def setName(self,name):
    self.__name__=name
  def getName(self):
    return self.__name__
  def getID(self):
    return self.__id__

stud1 = Student("John")
stud2 = Student()
stud3 = Student("June")

print( stud2.getName()) # Name unknown
print( stud3.getName()) # June
```

Operator overloading

We can overload standard operators applied to user-defined classes like this:

```
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
```

This defines the special method __add__, which is called when + is used with class instances.

See

https://docs.python.org/3/reference/datamodel.html?highlight=overloading#emulating-numeric-types

Try this out.
Extend it so you can say v3=v1-v2

Inheritance

This is done by syntax like

```
class Subclass(BaseClass)
```

Then Subclass inherits the members of BaseClass, and these can be over-ridden:

```
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):  # special method, used by print
    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())  # inherited method
print(stud3)            # Student: name=June
```

Examples

Random numbers

True random numbers are impossible using conventional hardware.
Instead we use pseudo-random numbers.

A common way to do this is to have an integer sequence x_n calculated as:

$x_n = (x_{n-1} * b + c) \bmod m$

with b c and m some constants. The initial value is called the *seed*.

```
"linear congruential random number generator"
# Standard way to make random numbers.
# Python has a module named random to do this - here we roll our own..

import time # needed for time()
# Seed from clock. Get a start value from the system clock. This
# means we will get different sequences on different runs
rngValue=int(time.time())

def random():
  # return a pseudo random integer.
  global rngValue # use the global variable, not a local rngValue
  # these work simply next number = last *b + c
  # values for b and c need to be chosen carefully, or get a seqeunce
  # which repeats quickly
  rngValue=rngValue*6364136223846793005+1442695040888963407
  # these values are from Donald Knuth
  # https://www-cs-faculty.stanford.edu/~knuth/
  return rngValue

def randInt(limit):
  # in range 0 to limit-1
  return random() % limit

for count in range(0,200): # show some
  print(randInt(1000), end=" ") # end=" " : put space between, not newline
```

Exceptions

For example - getting user input, and dividing with it. If they enter 0 we get overflow. Catch the exception and output a message. Repeat until there is no problem:

```
 # exceptions - invalid user 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")
```

Files records and fields

```
# text files
# write file
myFile = open("demo.txt", "w")# w = write to the file
myFile.write("ABC 123")
myFile.close()
# read it back:
myFile = open("demo.txt", "r")
str=myFile.read(3) # read just 3 characters
print(str) # ABC
myFile.close()
```

Queues

```
# queue
# Python has a queue module, but we write our own to show
# an ADT implementation
# We use OOP and base it on a list

class MyQueue:
  def __init__(self):
    # Construct an empty queue
    # The class wraps a list, which we want to be private,
    # so name it __list
    self.__list = []

  def get(self):
    if len(self.__list)==0:
      return None
    return self.__list.pop(0)


  def put(self, data):
    self.__list.append(data)
    return

# test it
q=MyQueue()
q.put(1)
q.put(2)
q.put(3)
print(q.get())
print(q.get())
print(q.get())
print(q.get())
```

Priority queue

```
# priority queue
# values are held in a queue in order of decreasing priority
# first out is item with highest priority


class MyPriorityQueue:
  def __init__(self):
    # Construct an empty queue
```

```
    self.__list = []

  def get(self):
    if len(self.__list)==0:
      return None
    return self.__list.pop(0)


  def put(self, data):
    # data consists of a list with 2 fields - [data value, priority]
    if len(self.__list)==0: # adding to empty q
      self.__list.insert(0,data)
      return
    priority=data[1]
    index=0
    notFound=True
    while notFound:
      if index==len(self.__list): # reached the end
        break
      if self.__list[index][1] > priority: # first place lower priority
        index+=1
      else:
        notFound=False
    self.__list.insert(index, data)
    return

  def notEmpty(self):
    return len(self.__list)!=0

# test it
q=MyPriorityQueue()
q.put(['one',5])
q.put(['two',1])
q.put(['three', 10])
q.put(['four', 0])
q.put(['five', 20])
while q.notEmpty():
  print(q.get())
```

Circular queue

```
# circular queue
# aka a ring buffer

class CircQ:
  def __init__(self):
    # the q wraps a list with a fixed size
    self.SIZE = 1000
    self.storage = [0] * self.SIZE
    self.tail = 0
    # tail is where the next item to REMOVE is
    # unless the q is empty
    self.head = 0
    # head is where the next item to INSERT is
    # so contents at head currently garbage
    self.empty = True
    self.full = False

  def put(self, data):
    if self.full == True:
      return None
    self.storage[self.head] = data
```

```
    self.head += 1
    self.empty = False
    if self.head == self.SIZE: # wrap around
      self.head = 0
    # check for full - two possibilities..
    # we've wrapped around, and head caught up with tail
    if self.head == self.tail:
      self.full = True
    # or tail still at 0 and head reached end of list
    if self.head == 0 and self.tail == 0:
      self.full = True

  def get(self):
    if self.empty == True:
      return None
    value = self.storage[self.tail]
    self.tail += 1
    if self.tail == self.SIZE: # also wraps around
      self.tail = 0
    if self.tail == self.head: # if it was the last item
      self.empty = True
    return value

  def isFull(self):
    return self.full

  def isEmpty(self):
    return self.empty



myCircQ = CircQ()
myCircQ.put(48)
myCircQ.put(21)
myCircQ.put(99)
myCircQ.put(99)

while not myCircQ.isEmpty():
 print(myCircQ.get())
```

Stacks

```
# stack
# We use OOP and base it on a list

class MyStack:
  def __init__(self):
    # Construct an empty stack
    # The class wraps a list, which we want to be private,
    # so name it __list
    self.__list = []

  def pop(self):
    if len(self.__list)==0:
      return None
    return self.__list.pop(0) # remove from head


  def push(self, data):
    self.__list.insert(0,data) # insert at head of list
    return
  def isEmpty(self):
```

```
    return len(self.__list) == 0

# test it
q=MyStack()
q.push(1)
q.push(2)
q.push(3)
while not q.isEmpty():
  print(q.pop()) # 3 2 1
```

Hash tables

```
# hash table

class HashTable:
  def __init__(self,n): # table has n buckets
    self.EMPTY=[0,0]    # each bucket is a list - (key, value) pairs
    self.SIZE=n
    self.buckets=[]     # empty list
    for n in range(0,n): # add empty buckets
      self.buckets.append(self.EMPTY)


  def hash(self, key):
    return key % self.SIZE  # very simple hash - range 0 to SIZE-1

  def put(self, key, value):
    where = self.hash(key)  # initial location
    print(key,"hashes to",where)
    if self.buckets[where]==self.EMPTY or self.buckets[where][0]==key:
      # if empty, put it there - or update value of existing key
      self.buckets[where]=[key, value]
      print("Inserted at",where)
      return
    else:
      print("Collision")
    while self.buckets[where]!=self.EMPTY and self.buckets[where][0]!=key: # look for
      where = (where+1)% self.SIZE         # next empty slot, wrapping round
    self.buckets[where]=[key, value]       # and put it there
    print("Inserted at",where)
    return

  def get(self, key):
    where = self.hash(key)                # initial location
    if self.buckets[where]==self.EMPTY:   # if empty, must be absent
      return None
    if self.buckets[where][0]==key:       # maybe found it
      return self.buckets[where][1]
    while self.buckets[where][0]!=key:    # or ha sbeen collision
      where = (where+1)% self.SIZE        # start linear search
      if self.buckets[where]==self.EMPTY: # to find it
        return None
    return self.buckets[where][1]



myTable=HashTable(1000)

# ordinary put
myTable.put(38677,"one")
myTable.put(747464,"two")
myTable.put(8377623,"three")
```

```
# and gets
print(myTable.get(38677)) # one
print(myTable.get(747464)) # two
# check not present
print(myTable.get(8677))  # None
# make collision
myTable.put(47464,"four")
# and get of collision
print(myTable.get(47464)) # four
# check over-write key
myTable.put(747464,"new")
print(myTable.get(747464)) # new
```

Linear search

```
# linear search in Python

# test input data
data=[3,8,12,7,5,4,9,28,-8,39]
target=5
index=0
while index<10: # 0 to 9
  if data[index]==target:
     break # break out of list if found
  index=index+1

# search ended
if data[index]== target:
  print("Found at index=",index)
else:
  print("Target not in list")
```

Binary search

```

# binary search

# test ordered data
data=[1,3,6,7,9,10,13,16,18,21,25,29,32,36,42,54,56,67,78,79,82]
target=56

low=0 # initialise range low to high
high=len(data)-1
index=(high+low) // 2 # middle. // is integer division
while True: # endless loop
  print(low, index, high) # to follow what happens
  if low>high:
     break # target not present
  if data[index]==target:
     break # target found
  if data[index]<target:
     low=index+1 # switch to upper half
  else:
     high=index-1 # lower half
  index=(high+low) // 2


# search ended
if data[index]== target:
  print("Found at index=",index)
```

```
else:
  print("Target not in list")
```

Bubble sort

```
# bubble sort

def bubble(list):
  "Do a bubble sort on the list"
  n=len(list)
  for count in range(1,n): # n times
     for index in range(0, n-1): # goes 0 to n-2
        if list[index] > list[index+1]: # we access index+1 -
           # goes to n-1 the final element is at index n-1
           temp=list[index] # do the swap here
           list[index]=list[index+1]
           list[index+1]=temp
  return list

myList=[6,2,7,3,4]
myList=bubble(myList)
print(myList)
```

Merge sorted lists

```
# merge two sorted lists

def merge(list1, list2):
  "Merge 2 sorted lists and return result"
  result=[] # initialise a local variable
  # until one list is emptied..
  while len(list1)!=0 and len(list2)!=0:
  if list1[0]<list2[0]: # if list1 head is smaller..
     # remove and add to result
     result.append(list1.pop(0))
  else: # do the same for list2
     result.append(list2.pop(0))
  if len(list1)==0: # list1 emptied first
     result.extend(list2) # copy all list2 to output
  else: # the other one
     result.extend(list1)
  return result

myList=merge([1,3,6,8,9], [0,3,7,9,10,11])
Binary Search Tree - in-order traversalprint(myList)
```

Mergesort

```
# mergesort

def merge(list1, list2):
.. as in other example

def mergeSort(list):
  res=[]
  listLen=1
  while listLen< len(list):
     newList=[] # used to hold merged sublists
     while (len(list)>0):
        l1=list[0:listLen] # get first part of list into l1
```

```
      list=list[listLen:] # and lose it
      l2=list[0:listLen] # same into l2
      list=list[listLen:]
      newList.extend(merge(l1,l2)) # put at end of newList
    list=newList # put back to list
    listLen*=2
  return list

list=[56,2,6,3,4,78,99,32,14]
list=mergeSort(list)
print(list) # [2, 3, 4, 6, 14, 32, 56, 78, 99]
```

Binary Search Tree - in-order traversal

```
class TreeNode():

  "A node in a binary tree. Each node has a key field and a value field"

  def __init__(self, keyParam, valueParam):

    self.left = None
    self.right = None
    self.key = keyParam
    self.value = valueParam

class BST(object):


  "A binary search tree class. Nodes have type TreeNode"

  def __init__(self):
    "Construct an new empty tree"
    self.root = None

  def insert(self, key, value):
    "Insert a key value pair in the tree"
    node = TreeNode(key, value)  # make a new node with this data
    where=None
    if self.root == None:
      self.root = node
      return
    else:  # find where to put it
      where = self.root
    while True:
      if where.key < key:  # want to go right
        if where.right == None:  # found space
          where.right = node  # put it there
          return  # and finish
        else:  # go right and continue
          where = where.right
      else:  # want to go left, in same way
        if where.left == None:
          where.left = node
          return
        else:
          where = where.left

  def traverse(self):
    "An in-order tree traversal"
```

```
    self.inOrder(self.root)

  def inOrder(self, where):
    "Visit the left sub-tree, the ode, and the right sub-tree"
    if where.left != None:
      self.inOrder(where.left)  # recurse left
    print(where.key, where.value)  # output data
    if where.right != None:  # recurse right
      self.inOrder(where.right)

  def search(self, key):
    "Search the tree for this key"
    where = self.root
    while True:
      if where.key == key:
        return where.value
      if where.key > key:
        where = where.left
      else:
        where = where.right
      if where == None:
        return "Not present"
# End of BST class

myTree = BST()
myTree.insert(50, "Joe")
myTree.insert(25, "Jim")
myTree.insert(75, "Jake")
myTree.insert(5, "Jack")
myTree.insert(26, "Joan")
myTree.insert(76, "June")

myTree.traverse()
# 5 Jack
# 25 Jim
# 26 Joan
# 50 Joe
# 75 Jake
# 76 June

print(myTree.search(26)) # Joan
print(myTree.search(27)) # Not present
```

Tree copy - pre-order traversal

```
# BST

class TreeNode(object):
  "A node in a binary tree. Each node has a key field and a value field"

  def __init__(self, keyParam, valueParam):
     self.left=None
     self.right=None
     self.key=keyParam
     self.value=valueParam

class BST(object):
  "A binary search tree class. Nodes have type TreeNode"

  def __init__(self):
     "Construct an new empty tree"
     self.root=None
```

```
  def insert(self, key,value):
     "Insert a key value pair in the tree"
     node=TreeNode(key,value) # make a new node with this data
     if self.root==None:
        self.root=node
        return
     else: # find where to put it
        where=self.root
     while True:
        if where.key < key: # want to go right
           if where.right==None: # found space
              where.right=node # put it there
              return # and finish
           else: # go right and continue
              where=where.right
        else: # want to go left, in same way
           if where.left==None:
              where.left=node
              return
           else:
              where=where.left

  def inOrderTraverse(self):
     "An in-order tree traversal"
     self.inOrder(self.root)

  def inOrder(self, where):
     "Visit the left sub-tree, the ode, and the right sub-tree"
     if where.left!=None:
        self.inOrder(where.left) # recurse left
     print(where.key, where.value) # output data
     if where.right != None: # recurse right
        self.inOrder(where.right)

  def preOrderTraverse(self):
     "A pre-order tree traversal"
     self.preOrder(self.root)

  def preOrder(self, where):
     "Visit the left sub-tree, the node, and the right sub-tree"
     print(where.key, where.value) # output data
     if where.left!=None:
        self.preOrder(where.left) # recurse left
     if where.right != None: # recurse right
        self.preOrder(where.right)


  def search(self, key):
     "Search the tree for this key"
     where=self.root
     while True:
        if where.key==key:
           return where.value
        if where.key>key:
           where=where.left
        else:
           where=where.right
        if where==None:
           return "Not present"
# End of BST class
```

```
def copy(tree):
  newTree=BST();
  newTree.root=clone(tree.root)
  return newTree

def clone(node):
  if node==None:
     return None
  newNode=TreeNode(node.key, node.value)
  newNode.left=clone(node.left)
  newNode.right=clone(node.right)
  return newNode


myTree=BST()
myTree.insert(50,"Joe")
myTree.insert(25, "Jim")
myTree.insert(75,"Jake")
myTree.insert(5,"Jack")
myTree.insert(26, "Joan")
myTree.insert(76,"June")

myTree.inOrderTraverse()
print("--------------------")
copyTree=copy(myTree)
copyTree.inOrderTraverse()
```

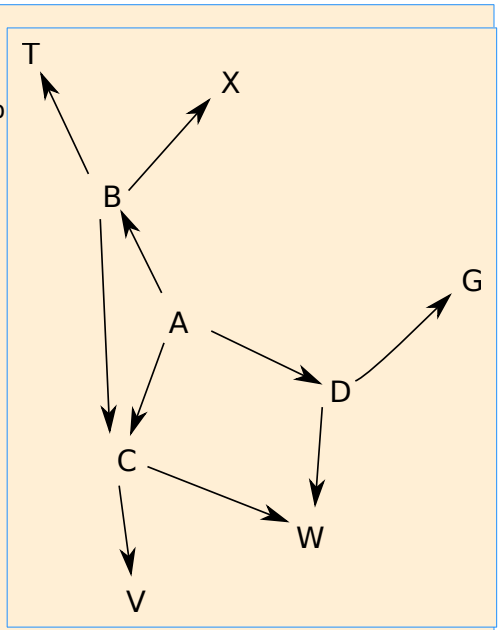
Graph - adjacancy list, breadth-first traversal

The code relates to the directed unweighted graph shown:

```
"graph - adjacency list - breadth-first traversal"

# set up data representing the graph
rowA = ['B','C', 'D'] # list of nodes A is linked to
rowB = ['C', 'T', 'X'] # nodes B is linked to
rowC = ['V', 'W'] # and so on
rowD = ['W', 'G']
rowG =[]
rowW =[]
rowV =[]
rowT =[]
rowX=[]
adjList = { # this is a dictionary - to make access
easy
'A': rowA,
'B': rowB,
'C': rowC,
'D': rowD,
'G': rowG,
'W': rowW,
'V': rowV,
'T': rowT,
'X': rowX
}

# show nodes linked to B:
whichNode='B'
list=adjList[whichNode]
```



```
for n in list:
  print(n)

# breadth-first traversal starting at A
nodeQ=[]
nodeQ.extend('A')
while not len(nodeQ)==0:
  node=nodeQ[0] # get q head
  nodeQ=nodeQ[1:] # and remove
  print("Visited ",node) # visit
  nextList = adjList[node] # get neighbours
  for n in nextList : # for each one
     if n not in nodeQ: # if not already in, add to q
        nodeQ.extend(n)
```

Reverse Polish

```
# infix to rpn

# set up
prec= { # dictionary of precedence levels of operators
  '+': 1,
  '-': 1,
  '*' : 2,
  '/': 2
  }
operands="0123456789"
stack=[] # main stack
opStack=[] # operator stack

infix='2+3*4';
while len(infix)!=0:
  token=infix[0]  # first char
  infix=infix[1:] # remove it
  if token in operands: # its an operand
    stack.append(token)
  else: # its an operator
    if len(opStack)==0 or prec.get(token)>prec.get(opStack[-1]):
      # higher precedence
      opStack.append(token)
    else: # lower
      op=opStack.pop()
      stack.append(op)
      opStack.append(token)
while len(opStack)!=0: # pop all opStack onto stack
  stack.append(opStack.pop())
print(stack)
```

Finite state machines

```
# FSM

# a parity machine
# make the machine
# combine next state and output tables
# each entry is a list (a,b) where a = next state, b = output
row0 = ((0, 0), (1, 1))
row1 = ((1, 1), (0, 0))
matrix = { # dictionary
  0: row0,
```

```
  1: row1
}

# run it with an input stream
inputList=(0,0,1,1,0,1,0,1,1)
state=0
for input in inputList:
  row=matrix.get(input)
  pair=row[state]
  output=pair[1]
  state=pair[0]
  print(input, output)
  ''' output:
0 0  even
0 0
1 1 odd
1 0 even..
0 0
1 1
0 1
1 0
1 1 '''
```

```
# FSM

# a serial binary adder
# make the machine
row0 = ((0, 0), (0, 1))
row1 = ((0, 1), (1, 0))
row2 = ((0, 1), (1, 0))
row3 = ((1, 0), (1, 1))
matrix = {
  "00": row0,
  "01": row1,
  "10": row2,
  "11": row3
}

# input matches 6+7
# least significant bits first
inputList=("01","11", "11","00")
state=0
for input in inputList:
  row=matrix.get(input)
  pair=row[state]
  output=pair[1]
  state=pair[0]
  print(input, output)
  ''' output is
01 1
11 0
11 1
00 1
= 13, least significant bit first  '''
```

```
# FSM

# an acceptor machine
# accept multiples of 2, in binary, only
```

```
row0 = (0,0)
row1 = (1,1)
table=(row0, row1)
startState=0
acceptState=0


# run it with an input stream
inputList=(1,1,0,0) # Binary number, msb first
state=0
for input in inputList:
  row=table[input]
  state=row[state]
if state==acceptState:
  print("Accept")
else:
  print("Reject")
```

Virtual methods and dynamic despatch

```
# OOP 3
# virtual methods and dynamic despatch

import random

class Base:
  def someMethod(self):
     print("Base")

class Sub(Base):
  def someMethod(self):
     print("Sub")




if random.random()>0.5:
  ref=Base()
else:
  ref=Sub()

ref.someMethod()  # is ref a Base or a Sub?
```

Regular expressions

```
# FSM as regular expression

def fsm(inputList):
  # an acceptor machine
  # accept 'ab' then anything
  # set up fsm
  # 3 input classes
  row0 = (1, 3, 2, 3)  # input a
  row1 = (3, 2, 2, 3)  # input b
  row2 = (3, 3, 2, 3)  # anything not a or b
  table = (row0, row1, row2)
  startState = 0
  acceptState = 2
  state = startState
  # go
  for input in inputList:
```

```
    if input == 'a':
      inputClass = 0
    else:
      if input == 'b':
        inputClass = 1
      else:
        inputClass = 2
    row = table[inputClass]
    state = row[state]
  if state == acceptState:
    return "Accept"
  else:
    return "Reject"


# run it
print(fsm(('a', 'b', 'a', 'a', 'x')))  # accept
print(fsm(('a', 'a', 'a', 'a', 'x')))  # reject
print(fsm(('b', 'b', 'a', 'a', 'x')))  # reject
print(fsm(('x', 'x', 'a', 'a', 'x')))  # reject
print(fsm(('a', 'b')))              # accept
```

```
# the Python regular expression module
# use of pre-compiled regex objects
import re

myRegExp = re.compile('bdf') # the string bdf
matchObject=myRegExp.match('bdfffbda') # match = match at start
print(matchObject) # <_sre.SRE_Match object; span=(0, 3), match='bdf'>
print(matchObject.start()) # 0
matchObject=myRegExp.match('fdboo')
print(matchObject) # None
searchObject=myRegExp.search('0123bdfxyz') # search all the way through
print(searchObject) # <_sre.SRE_Match object; span=(4, 7), match='bdf'>
print(searchObject.start()) # 4
```

Functional programming

```
# functional programming 1

# define a function
def average(a,b):
  return (a+b)/2

# now average is an object, with type 'function'
print(type(average)) # <class 'function'>
myFunction = average # now myFunction is a reference to this function object
# and we can apply it:
print(myFunction(2,3)) # 2.5
```

Pure functions

```
# pure functions

# define a function which is not pure
def impure(x):
  return x+b
```

```
b=3 # set state
print(impure(2)) # 5
b=6 # change state
print(impure(2)) # 8
# the output of an impure function depends on state

def pure(x):
  b=2 # b is local
  return b*x+1

print(pure(2)) # 5
b=5 # change state
print(pure(2)) # 5
# output of pure is the same, no matter what system state is

def f(x):
  global b
  b+=1
  return b*x+1

b=1
print(f(2)) # 5
# but f has a side-effect
print(b) # 2
```

Functions as parameters

```
# functional programming 3
# functions passed as parameters

# define 2 functions
def f(x):
  return 2*x+1

def g(x):
  return x*x

def apply(aFunction,x):
  return aFunction(x)

print(apply(f,2)) # 5
print(apply(g,2)) # 4

def compose(f1,f2,x):
  return f1(f2(x))

print(compose(g,g,2)) # 16 : g(2)=4 g(4)=16
print(compose(f,g,2)) # 9 : g(2)=4 f(4)=9
print(compose(g,f,2)) # 25 : f(2)=5 g(5)=25
```