Guide to Using Python 3

Computational Methods and Modelling 3

Edward McCarthy

September 2022

THE UNIVERSITY of EDINBURGH
School of Engineering

Iterative Loops in Python: for

There are a number of ways to programme repetitive or iterative operations in Python

```
for i = 1 to 10
<loop body>
```

Here, the body of the loop is executed ten times and then stops.

```
for (i = 1; i <= 10; i+-)
<loop body>
```

Here, i is initialized at 1, it must remain less than or equal to 10 (termination criterion), and the value of i increases by 1 after every loop

```
for n in (0, 1, 2, 3, 4): ...
print(n)
```

Here, the programme prints each element in the bracket list.

```
x = range(5)
for n in x:
print(n)
```

This is a more efficient way of defining an iteration range that has been defined once (x)

THE UNIVERSITY of EDINBURGH
School of Engineering

Iterative Loops in Python: while

There are a number of ways to programme repetitive or iterative operations in Python

```
i = 1
while i < 6:
  print(i)
  i += 1
```

The while command usually specifies a continuing condition for the duration of the loop such as less than (<) or greater than (>). Thus, the index (i) must be initialised before *while* is called.

While can be used with break: in this case the loop terminates if I = 3 (not a very useful loop structure; it just illustrates what will happen if break is used like this.)

It can also be used with continue (right): in this case the loop will continue beyond I = 3 until 6 as instructed in the while command. Again, illustrative, not practical.

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

This means increase the index, i by 1 unit

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

Iterative Loops in Python: break and continue

One must sometimes specify ways of breaking or continuing a loop that is subject to criteria such as in the following examples

break stops the loop and returns the first value that is valid, but no more terms

```
for i in [‘axe', 'bar', ‘back', ‘queen']: ...
if 'b' in i: ...
break ...
print(i)
..axe
```

continue stops the iteration it occurs in and starts the next iteration if a certain criterion(a) is/are satisfied (i.e. via an if statement).

```
for i in [‘axe', 'bar', ‘back', ‘queen']: ...
if 'b' in i: ...
continue ...
... axe queen
```

THE UNIVERSITY of EDINBURGH
School of Engineering

Iterative Loops in Python: for with else

Occasionally, a for statement contains an else clause that enables conditionality for the loop without explicitly using an if statement

```
for i in [‘axe', ‘bar',
‘back', ‘queen']: ...
print(i) ...
else: ...
print('Done.’)
... axe bar back queen
Done.
```

THE UNIVERSITY of EDINBURGH
School of Engineering

If Statements…Conditional Programming

The simple conditional 'if' statement has the following syntax in Python 3.7

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

(Note that the indent on 'print' is mandatory for Python.

To introduce a second condition we use the command **elif**

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

https://www.w3schools.com/python/python_conditions.asp

If Statements…Conditional Programming

To introduce a last condition (in this example a last, third condition) we use **else**

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Numerous intermediate **elif** statements can be made in the body of this type of a loop until the last else condition.

https://www.w3schools.com/python/python_conditions.asp

THE UNIVERSITY *of* EDINBURGH
School of Engineering

If Statements…Conditional Programming

A combined condition can also be specified using one if command and **‘and’**

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

The **or** statement can also be used in similar manner to give greater flexibility

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

https://www.w3schools.com/python/python_conditions.asp

If Statements…Conditional Programming

If statements can also be ***nested*** (one loop inside the other) as follows:

```
x = 41
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

Obviously, no real programme would write a statement like this where the value of x was both known and predetermined within the code; this is purely for illustration.

https://www.w3schools.com/python/python_conditions.asp

THE UNIVERSITY *of* EDINBURGH
School of Engineering

Importing Data in Python

1. **Importing and exporting data to and from Python** can be achieved efficiently using the following commands:
2. Firstly, if there is a text file (.txt) containing the data it can be imported as follows. Note that the print command is optional.

```
file = open("sample.txt")
data = file.read()
print(data)
file.close()
```

3. An alternative option is to use the **with** command to achieve the same importation: the essential difference is that the **with** command automatically closes the file after importing data.

```
with open("welcome.txt") as file:
object data = file.read()
```

4. Lastly, one may prompt the user to enter data using the **input** command:

```
value = input("Please enter a string:\n")
print(f'You entered {value}')
```

Useful Computational Tools in Python

1. There are two ways to use Python.
   a. **By use of explicit programming** using do loops and if-then-else statements (left panel on Spyder interface).
   b. **By use of inbuilt one-line commands** designed to perform standard tasks (i.e., root finding, integration and optimisation) quickly with minimum intervention and no coding effort. (lower right panel on Spyder interface)
2. It might seem that one would always choose to use b), but there will be occasions where a complex problem requires explicit and self-written coding solutions.

The next slide shows a variety of standard codes that have already written for the Python platform by various users over its years of development

Definition of Functions in Python

The definition of mathematical functions in Python:

Python has a number of conventions for defining a function.

```
def my_function(x):
  return 5 * x
```

To evaluate the function for various values of x, one can use the **print** command as below:

```
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

The second common convention for defining a function is the **lambda** function. Here, a function can be defined without a dedicated name.

```
x = lambda a, b : a * b
print(x(5, 6))
```

This convention has a number of advantages, i.e., lambda can be used repeatedly within a routine to define numerous different functions.

THE UNIVERSITY of EDINBURGH
School of Engineering

Symbolic Computation

Lastly, it is possible to do algebra electronically within Python using standard commands:

```
from sympy import *
x = Symbol('x') y = Symbol('y’)
print 2*x + 3*x - y # Algebraic computation
print diff(x**2, x) # Differentiates x**2 wrt. x
print integrate(cos(x), x) # Integrates cos(x) wrt. x
print simplify((x**2 + x**3)/x**2) # Simplifies
expression
print limit(sin(x)/x, x, 0) # Finds limit of sin(x)/x
as x->0
print solve(5*x - 15, x) # Solves 5*x = 15
```

Useful Computational Tools in Python: Standard Inbuilt Functions

Python (Solving Equations Algebraicly using Sympy package)

sympy for solving multiple equation systems for each variable.

Code

```
import sympy as sym
sym.init_printing()
x,y,z = sym.symbols('x,y,z')
c1 = sym.Symbol('c1')
f = sym.Eq(2*x**2+y+z,1)
g = sym.Eq(x+2*y+z,c1)
h = sym.Eq(-2*x+y,-z)

sym.solve([f,g,h],(x,y,z))
```

Objects to be solved

Solution outputs

Here comma indicates the '=' sign, i.e., $2x^2+y+z=1$

Result

$$x = -\frac{1}{2} + \frac{\sqrt{3}}{2}$$

$$y = c_1 - \frac{3\sqrt{3}}{2} + \frac{3}{2}$$

$$z = -c_1 - \frac{5}{2} + \frac{5\sqrt{3}}{2}$$

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Solving Equations)

fsolve for solving multiple equation systems (nonlinear equations)

```
import numpy as np
from scipy.optimize import fsolve

def myFunction(z):
   x = z[0]
   y = z[1]
   w = z[2]

   F = np.empty((3))
   F[0] = x**2+y**2-20
   F[1] = y - x**2
   F[2] = w + 5 - x*y
   return F

zGuess = np.array([1,1,1])
z = fsolve(myFunction,zGuess)
print(z)
```

Imports fsolve from standard scipy.optimise package

Creates an empty row array of 3 elements

Populates guess array for solution with three guess values (not necessarily known roots).

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Linear Algebra)

linalg for solving multiple equation systems (linear equations)

```
import numpy as np

A = np.array([ [3,-9], [2,4] ])
b = np.array([-42,2])
z = np.linalg.solve(A,b)
print(z)

M = np.array([ [1,-2,-1], [2,2,-
1], [-1,-1,2] ])
c = np.array([6,1,1])
y = np.linalg.solve(M,c)
print(y)
```

Creates a matrix row by row from the top.

linalg solves the equation immediately using a method such as Gauss Seidel or equivalent ‘in the background’.

Two separate examples given here.

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Numerical Integration)

Numerical integration of continuous functions over a certain domain

quad -- General purpose integration.

dblquad -- General purpose double integration.

tplquad -- General purpose triple integration.

fixed_quad -- Integrate func(x) using Gaussian quadrature of order n.

quadrature -- Integrate with given tolerance using Gaussian quadrature.

romberg -- Integrate func using Romberg integration. Methods for Integrating Functions given fixed samples.

trapz -- Use trapezoidal rule to compute integral from samples.

cumtrapz -- Use trapezoidal rule to cumulatively compute integral.

simps -- Use Simpson's rule to compute integral from samples.

romb -- Use Romberg Integration to compute integral from (2**k + 1) evenly-spaced samples. See the special module's orthogonal polynomials (special) for Gaussian quadrature roots and weights for other weighting factors and regions. Interface to numerical integrators of ODE systems.

odeint -- General integration of ordinary differential equations.

ode -- Integrate ODE using VODE and ZVODE routines.

THE UNIVERSITY *of* EDINBURGH
School of Engineering

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Numerical Integration)

quad Standard Function

Different parameters for the equation can be applied in the *args* argument. The example below calculates the below integral for specified a, b in the domain 0 < x < 1:

$$I(a,b)=\int_0^1 ax^2+b\,dx.$$

This integral can be evaluated by using the following code:

```
from scipy.integrate import quad
def integrand(x, a, b):
return a*x**2 + b
a = 2
b = 1
I = quad(integrand, 0, 1, args=(a,b))
I (1.6666666666666667, 1.8503717077085944e-14)
```

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Numerical Integration)

dblquad Standard Function

The **dblquad** function performs double integration on the same function in different variables where **lambda** defines the function

Example: the integral $$I = \int_{y=0}^{1/2} \int_{x=0}^{1-2y} xy \, dx \, dy = \frac{1}{96}.$$

```
from scipy.integrate import dblquad
area = dblquad(lambda x, y: x*y, 0, 0.5, lambda x: 0,
lambda x: 1-2*y)
area (0.010416666666666668, 1.1564823173178715e-16)
```

Note: the routine defines the outer integral with fixed bounds first, then uses the **lambda** command to define the non-fixed bound of the inner integral.

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Optimisation)

minimise for unconstrained multivariate optimisation

```
import numpy as np
from scipy.optimize import minimize
def rosen(x): """The Rosenbrock function"""
return (100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-
x[:-1])**2.0)
x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='nelder-
mead', options={'xatol': 1e-8, 'disp': True})
Optimization terminated successfully. Current
function value: 0.000000 Iterations: 339
Function evaluations: 571
print(res.x)
```

THE UNIVERSITY *of* EDINBURGH
School of Engineering

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Optimisation)

rosen hess for unconstrained multivariate optimisation using gradients

```
def rosen_hess(x):
x = np.asarray(x)
H = np.diag(-400*x[:-1],1) - np.diag(400*x[:-1],-1)
diagonal = np.zeros_like(x)
diagonal[0] = 1200*x[0]**2-400*x[1]+2
diagonal[-1] = 200
diagonal[1:-1] = 202 + 1200*x[1:-1]**2 - 400*x[2:]
H = H + np.diag(diagonal)
return H
res = minimize(rosen, x0, method='Newton-CG',jac=rosen_der,
hess=rosen_hess, options={'xtol': 1e-8, 'disp': True})
res.x array([1., 1., 1., 1., 1.])
```

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Optimisation)

nonlinearconstraint for **constrained** multivariate optimisation using gradients

```
def cons_f(x):
return [x[0]**2 + x[1], x[0]**2 - x[1]]
def cons_J(x):
return [[2*x[0], 1], [2*x[0], -1]]
def cons_H(x, v):
return v[0]*np.array([[2, 0], [0, 0]]) +
v[1]*np.array([[2, 0], [0, 0]])
from scipy.optimize import NonlinearConstraint
nonlinear_constraint = NonlinearConstraint(cons_f, -
np.inf, 1, jac=cons_J, hess=cons_H)
```

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Optimisation)

for **constrained** multivariate optimisation using gradients

For the previous codes, boundaries and linear constraints for the optimisation routine are specified with commands such as these.

```
from scipy.optimize import Bounds >>>
bounds = Bounds([0, -0.5], [1.0, 2.0])
```

```
from scipy.optimize import LinearConstraint >>>
linear_constraint = LinearConstraint([[1, 2],
[2, 1]], [-np.inf, 1], [1, 1])
```

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (Optimisation)

The final solution of a constrained non-linear system is given in the following example where all the arguments of the minimise function have been defined:

```
x0 = np.array([0.5, 0])
res = minimize(rosen, x0, method='trust-constr',
jac=rosen_der, hess=rosen_hess,
constraints=[linear_constraint,nonlinear_constraint],
options={'verbose': 1}, bounds=bounds)
# may vary `gtol` termination condition is satisfied.
Number of iterations: 12, function evaluations: 8, CG
iterations: 7, optimality: 2.99e-09, constraint
violation: 1.11e-16, execution time: 0.016 s. >>>
(res.x) [0.41494531 0.17010937]
```

THE UNIVERSITY of EDINBURGH
School of Engineering

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (ODE Solvers)

The solution of ODEs in Python is supported by **odeint** from the Scipy package:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# function that returns dy/dt
def model(y,t):
    k = 0.3
    dydt = -k * y
    return dydt
# initial condition
y0 = 5
# time points
t = np.linspace(0,20)

# solve ODE
y = odeint(model,y0,t)
# plot results
plt.plot(t,y)
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()
```

This script solves the equation:

$$\frac{dy(t)}{dt} = -k\, y(t)$$

https://apmonitor.com/pdc/index.php/Main/SolveDifferentialEquations

Useful Computational Tools in Python: Standard Inbuilt Functions

Python Numerical Analysis Standard Functions (ODE Solvers)

A family of solutions can be solved for different values of the parameter, k as below:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# function that returns dy/dt
def model(y,t,k):
    dydt = -k * y
    return dydt

# initial condition
y0 = 5
# time points
t = np.linspace(0,20)
# solve ODEs
k = 0.1
y1 = odeint(model,y0,t,args=(k,))
k = 0.2
y2 = odeint(model,y0,t,args=(k,))
k = 0.5
y3 = odeint(model,y0,t,args=(k,))
```

```
# plot results
plt.plot(t,y1,'r-',linewidth=2,label='k=0.1')
plt.plot(t,y2,'b--',linewidth=2,label='k=0.2')
plt.plot(t,y3,'g:',linewidth=2,label='k=0.5')
plt.xlabel('time')
plt.ylabel('y(t)')
plt.legend()
plt.show()
```

This script solves the same equation repeatedly for different values of k (parameterisation):

$$\frac{dy(t)}{dt} = -k\, y(t)$$

https://apmonitor.com/pdc/index.php/Main/SolveDifferentialEquations

THE UNIVERSITY of EDINBURGH
School of Engineering

Summary of Python Capability for General Numerical Methods

- The structure of iterative (do, for, while) and conditional (if, elif, else) statements in Python has been introduced and explained.
- Input of data into Python from files and via user input to the command window has been outlined.
- Use of Python functions for solution of algebraic equations using custom package **sympy** has been demonstrated.
- Python, like commercial programming languages such as MATLAB, has its own standard toolboxes or packages that feature pre-written programmes for a wide range of numerical operations and routines
- **numpy** is written to enable a wide range of linear algebra techniques to be applied, while **scipy** is used to perform numerical optimisations. (single and univariate; constrained (with conditions) and unconstrained (simple function minimisations and maximisations).
- The Python open-source, online community continues to develop these tools, and you can become one of these developers!