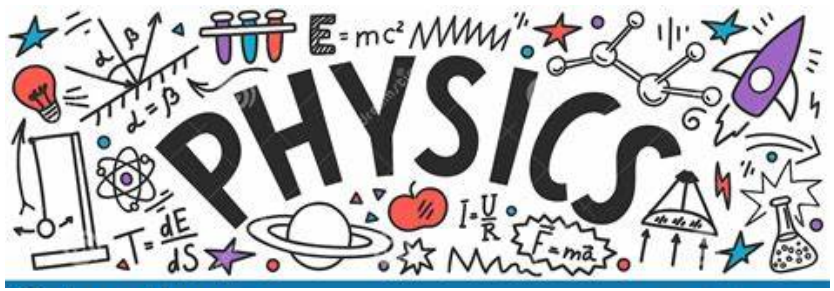
Python Programming for Physicists

Bin Gu

gubit@163.com
Department of Physics
Nanjing University of Information Science and Technology

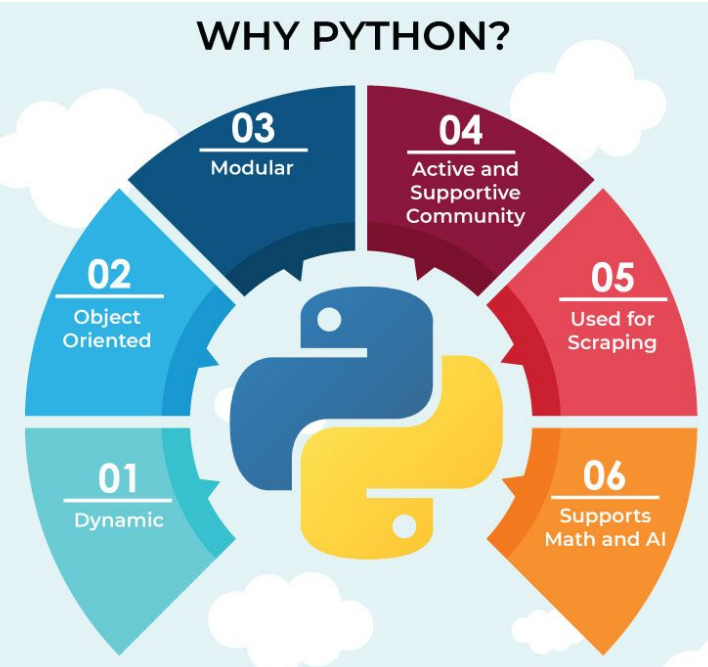
September 21, 2023

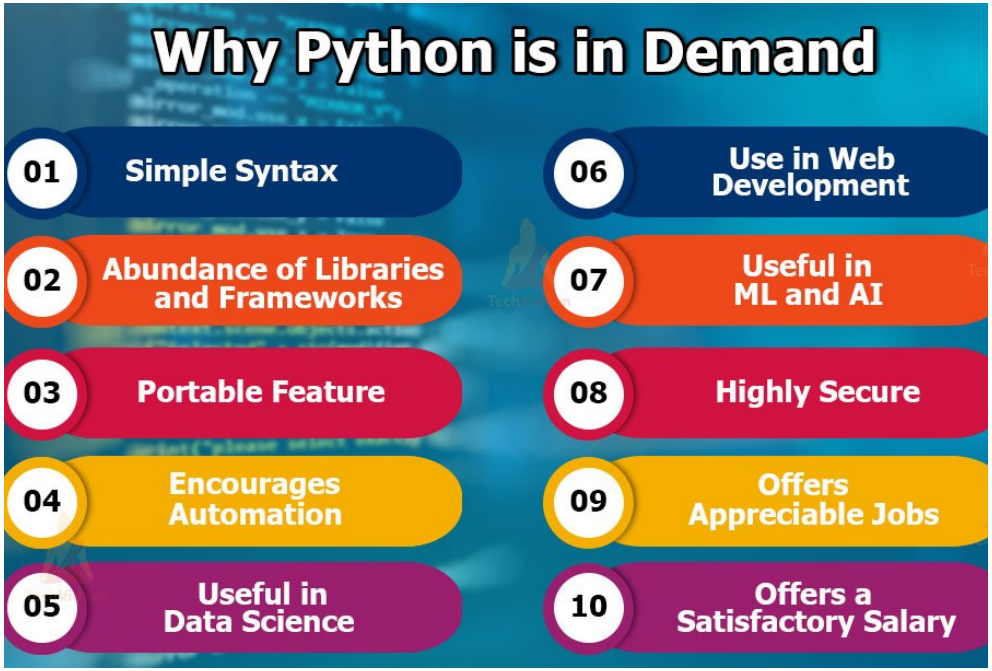


1. Background: Why python?
2. Getting started
3. Basic Python Syntax
4. Flow Controlling
5. LIST and ARRAY
6. "For" LOOPS
7. User defined functions
8. Good Style

1. Background: Why python ?

https://www.python.org





Python is a high-level free programming language that lets you work more quickly and integrate your systems more effectively.

For Computational physics:

1. flexible data types:
complex numbers, vectors, matrices, tensors, sets, $\cdots$

2. professional build-in facilities:
constants, matrix operations, Fourier transforms, $\cdots$

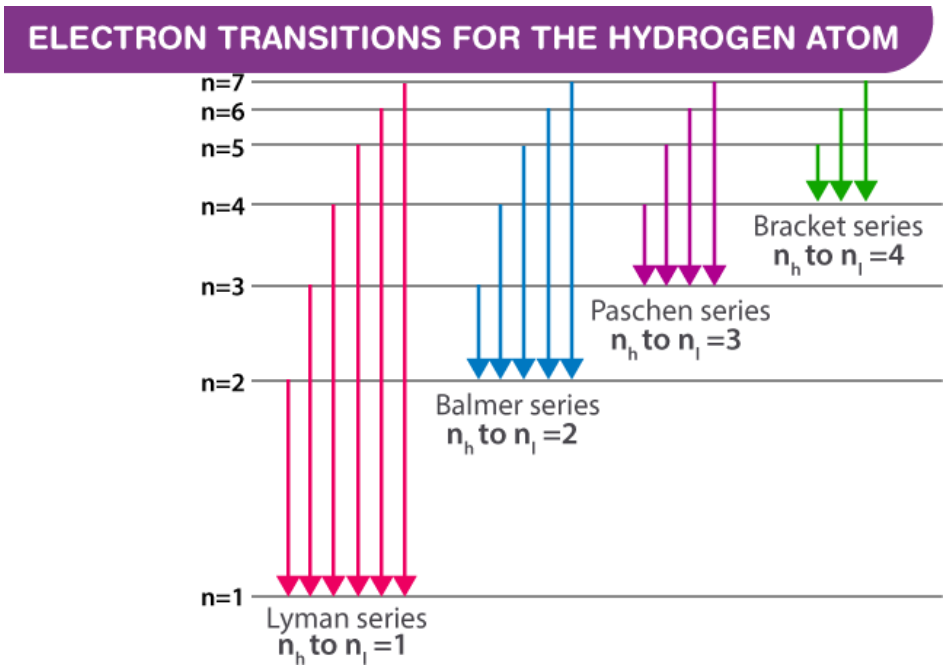
3. versatile visualization: diagrams, models, animations, $\cdots$

An example

In 1888 Johannes Rydberg published his famous formula for the wavelengths λ of the emission lines of the hydrogen atom:

$$\frac{1}{\lambda} = R(\frac{1}{n_l^2} - \frac{1}{n_h^2})$$

in which, $n_h > n_l$. Question: Calculate the wavelengths of the first five lines in each of these three series.



Solution:

1. Write the Python code with a **text editor**, and save it with the name: rydberg.py

```
R = 1.097e-2
for m in [1,2,3]:
    print("Series for m =",m)
    for k in [1,2,3,4,5]:
        n = m + k
        invlambda = R*(1/m**2-1/n**2)
        print(" ", 1/invlambda, "nm")
```

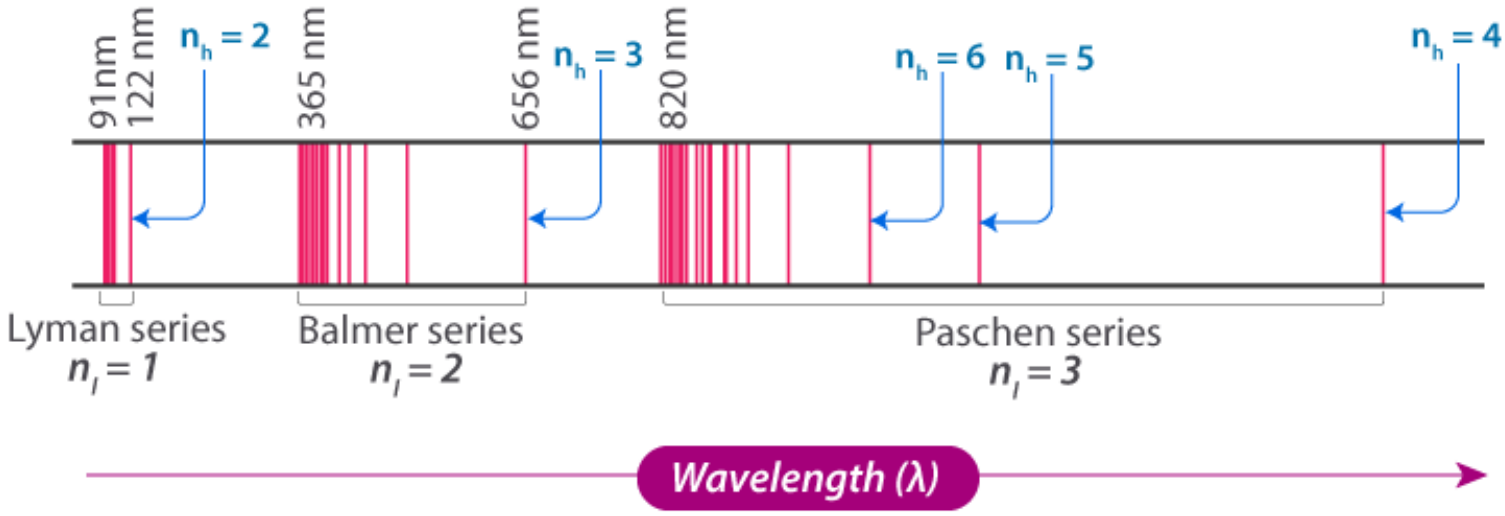
2. Run python

```
bingu@T480s:~$ python3 rydberg.py
```

3. Results

```
Series for m = 1
  121.5436037678517 nm
  102.55241567912488 nm
  97.23488301428137 nm
  94.95594044363415 nm
  93.76220862091418 nm
Series for m = 2
  656.3354603463993 nm
  486.1744150714068 nm
  434.084299170899 nm
  410.2096627164995 nm
  397.04243897498225 nm
Series for m = 3
  1875.2441724182836 nm
  1281.9051959890612 nm
  1093.8924339106654 nm
  1005.013673655424 nm
  954.6697605038536 nm
```

4. Visualization



In addition to Python

Python is not everyone's choice of programming language for every task.

There are many other programming languages available and there are tasks for which another choice may be appropriate.

If you get serious about computational physics you are, without doubt, going to need to write programs in another language someday. Common choices for scientific work (other than Python) include C, C++, Fortran, and Java.

All of the languages above use basically the same concepts, and differ only in relatively small details of how you give specific commands to the computer.

2. Getting started

Learn Python online: pynative.com geeksforgeeks.org

Installing Python3 and packages required:

the Python language itself, and the packages "numpy", "matplotlib", and "visual" (also called "VPython" in some places).

→ www.python.org
→ www.vpython.org
→ matplotlib.org/downloads.html
→ sourceforge.net/projects/numpy/files/NumPy

Note: difference between Python2 and Python3.

Install numpy package, using Pip.

Pip is a command-line tool that allows you to install, uninstall and upgrade software packages written in Python.

```
python -m pip --version
# check the version of pip

# Windows:
# download *.py from :https://pypi.org/project/pip/#downloads
python get-pip.py

# Linux:
sudo add-apt-repository universe
sudo apt install python3-pip
```

Numpy:

```
# Windows:
download *.whl from http://mirrors.aliyun.com/pypi/simple/numpy/
# version is consistent with python3.x. whl
cd C:\Users\bingu\AppData\Local\Programs\Python\Python310
pip install .\numpy-1.21.3-cp310-cp310-win_amd64.whl

# Linux
python3.8 -m pip install numpy
python3.8 -m pip install numpy --index -url https://pypi.tuna.tsinghua.edu.cn/simple
# add ~/.loacal/bin to PATH
```

3. Basic Python Syntax

- **Variable**

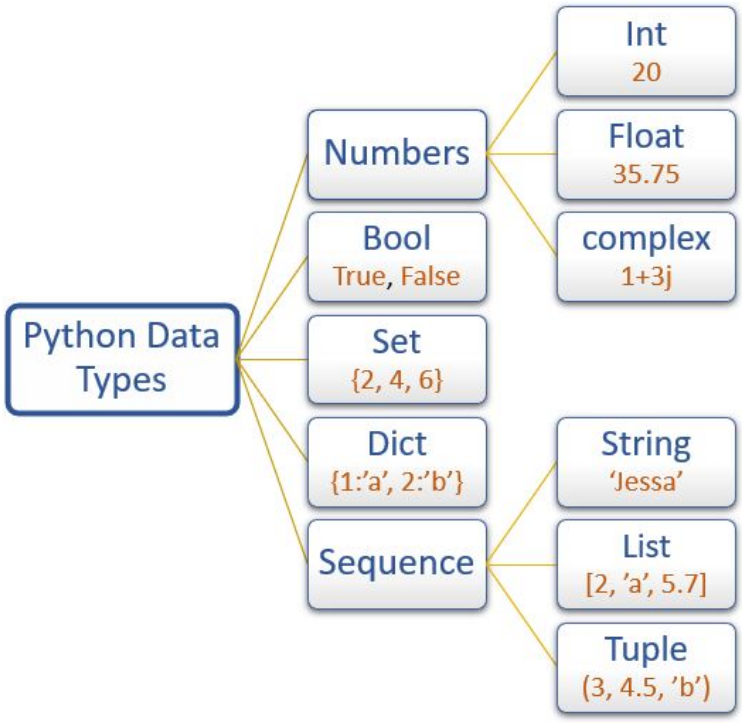
Q2.1: What's a ***variable***?

A **variable** is created the moment we first assign a value to it.

The **type of a variable** is set by the value that we give it, and can change as a Python program runs.

```
x=1
y=1.0
z=float(1)  #specify the data type of a variable, casting
print(type(z)) #get the data type of z with the \texttt{type()} function

var1 = [5,"a", 3+2.5j]
print(type(var1))
del var1
print(var1) #NameError: name 'var1' is not defined

roll_no, marks, name = 10, 70, "Jessa" #multiple assignments
```

- **Data Types**




```
Q_ELE=1.6e-19 # UPCASES constants

import constant.py # A module to define "
    unchanged" variables.
```

Tip: Always keep the type of a variable in a /given scope/ of the program.
Variable is case-sensitive.
There is no "Constant" (type).
Import constant module.

Question 2.1: Couldn't all values, including integers and real numbers, be represented with complex variables, so that we only need one type of variable?

- **ARITHMETIC**

x+y	addition
x-y	subtraction
x*y	multiplication
x/y	division
x**y	raising x to the power of y.
x//y	integer division rounded down to the nearest integer of int(x)/y. e.g.: -14//3 gives -5.
x%y	modulo, which means the remainder after xis divided by y. e.g.: n%2 is zero if n is even (and one if n is odd).

```
x = a + 2*b - 0.5*(1.618**c + 2/7)
print (a + b/c)
x = x**2+2
#evaluate the value of the right-
      hand side of =, then set x to
      this new value.
x,y = 2*z+1,(x+y)/3
# calculate both the values, from
      the current x, y, and z.
x,y = y,x
# Question: what will Python do?
```

Note: Generally, the end result of arithmetic operation has the more general of the two types that went into the operation. Exception: "/" division always gives floating-point or complex values.

Modifiers

```
#add 1 to x (i.e., make x bigger by 1)
x += 1
#subtract 4 from x
x -= 4
#multiply x by -2.6
x *= -2.6
#divide x by 5 times y
x /= 5*y
#divide x by 3.4 and round down to an integer
x //= 3.4
```

COMMENTS: in Python any program line that starts with a hash mark "#" is ignored completely by the computer, and thus is looked as comments / reminds by the programmer.

- **Functions, Modules, Packages, Library**

Function The function is a block of code defined with a name.
We use functions whenever we need to perform the same task multiple times.
DRY principle: Don't Repeat Yourself.
Re-usability and modularity, and hence, improve efficiency and reduce errors.

A Function can take arguments and returns the value.

- Built-in function:
  `range()`, `id()`, `type()`, `input()`, `eval()` etc.

- User-defined function:

```
def function_name(parameter1, parameter2):
        # function body
        # write some action
return  value
```

Module A module is a Python file that contains **collections of functions and global variables**, and with a name having `.py` extension.

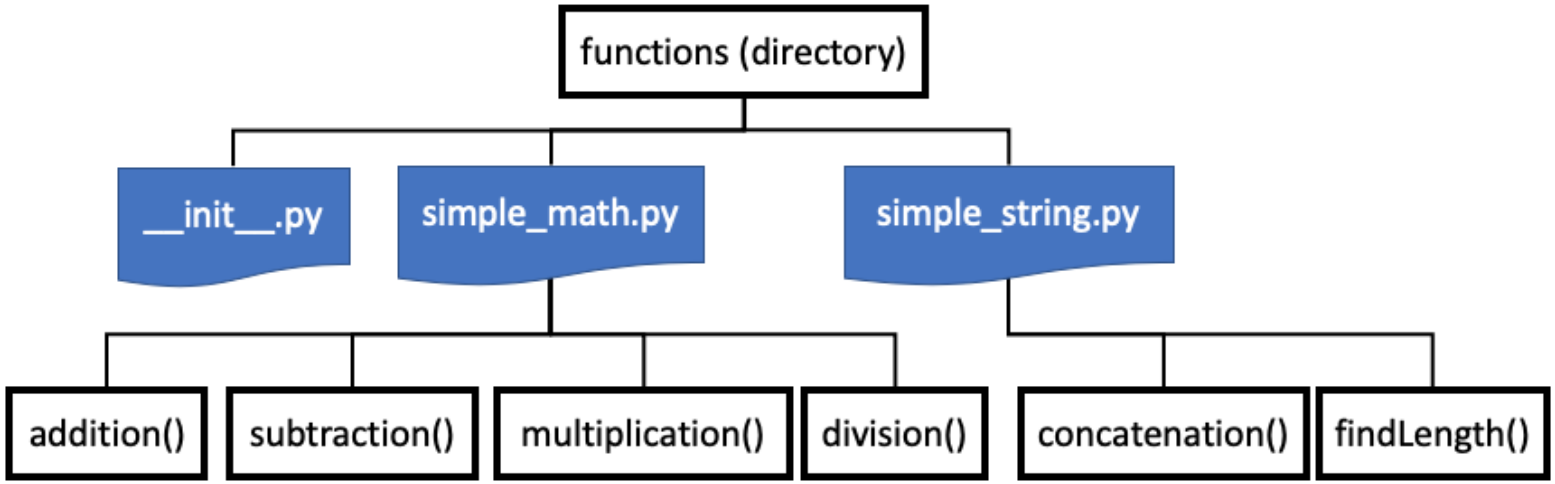
Package A package is a simple directory having collections of modules, and a `__init__.py` file by which the interpreter interprets it as a Package or sub-package.

NumPy is the fundamental Python package for scientific computing.

Library A Library is a collection of related functionality of codes (modules) that allows you to perform many tasks without writing your code.

`Matplotlib` library is a standard library for generating data visualizations in Python. It supports building basic two-dimensional graphs as well as more complex animated and interactive visualizations.

A typical structure of a package named "functions":



Note: library, package (and module) are often used interchangeably.

Example: The `math` package:

log, log10, exp, sqrt

sin, cos, tan, asin, acos, atan, sinh,cosh,tanh

import statement and the "dot" syntax

```
#"import" the logarithm function from the math package
from math import log
x = log(2.5)
#import a list of functions from a package
from math import log,exp,sin,cos,sqrt,pi,e
from math import * # Not advised

#sub-package
from numpy.linalg import inv #inverse of a matrix
import numpy.linalog.inv

#import numpy lib and rename it as np
import numpy as np
arr = np.array([1,2,4,16])
```

- **Interactive functions: PRINT and INPUT**

`print()` print statement. Always prints the current value of the variable at the moment the statement is executed.

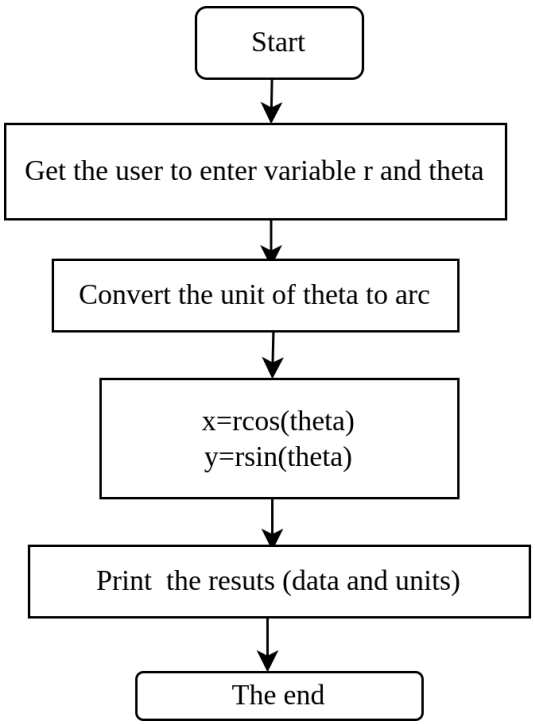
```
x = 1
y = 2
z=2+3j
print(x)
print(x,z,sep=" . .. ")
print("The value of x is",x, "and the value of y is",y)
```

`input()` input(prompt), the statement prints out the quantity of prompt (if any) inside the parentheses. Then, wait the user to type a value (String) on the keyboard.

```
x = input()
y = input("please input y:")
print ("The value of y is" , y) #y is a string

x = float(input("Enter the value of x: "))
print("The value of x is",x) # x is a float number
```

Example 2.1 (P.34)

Suppose the position of a point in two-dimensional space is given in polar coordinates (r, θ) in units of meters and degrees individually. Please convert it to Cartesian coordinates (x, y) with Python.



- **define variable**
- **figure out calculations**

- import module/lib
- input()
- calculate /use functions
- print() / formated
- comments / readability
- idle / **.py and debug

coord.py:

```
# convert (r,theta) to (x,y)
from math import sin,cos,pi
# casting the input value to float-
    pointing
r = float(input("Enter r in m:"))
d = float(input("Enter theta in deg:"))
# degree to arc
theta = d*pi/180
x = r*cos(theta)
y = r*sin(theta)
# print into two lines
print("x = ",x,"m\ny = ",y,"m")
```

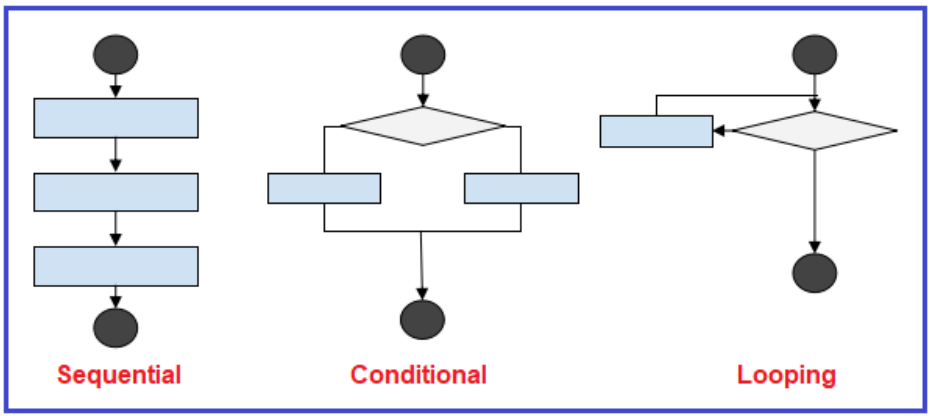
Homework

2.4 Traveling at a relativistic speed (p.36)

2.6 Planetary orbits (p.36)

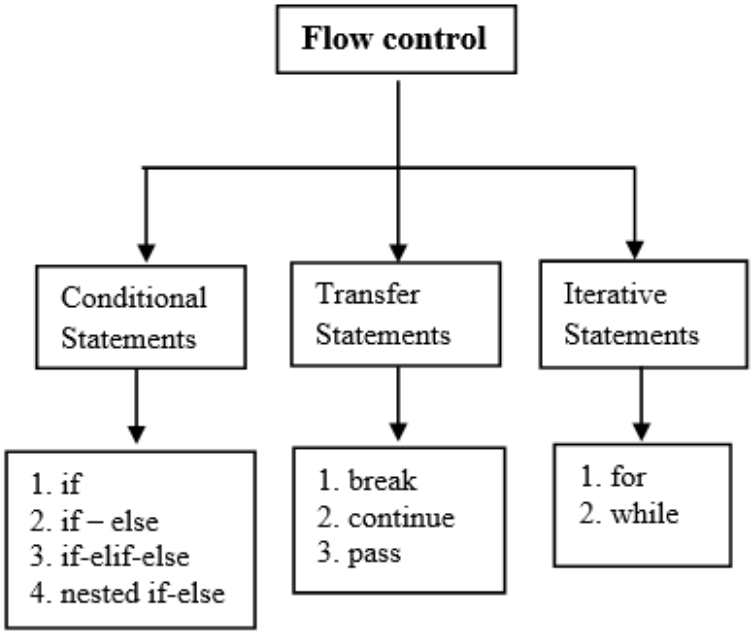
4. Flow Controlling

Instead of marching Sequentially from one statement to the next, a "smart" program should make decisions based on given criteria, so as to jump around the program.



Conditional: Statements are executed based on the condition.

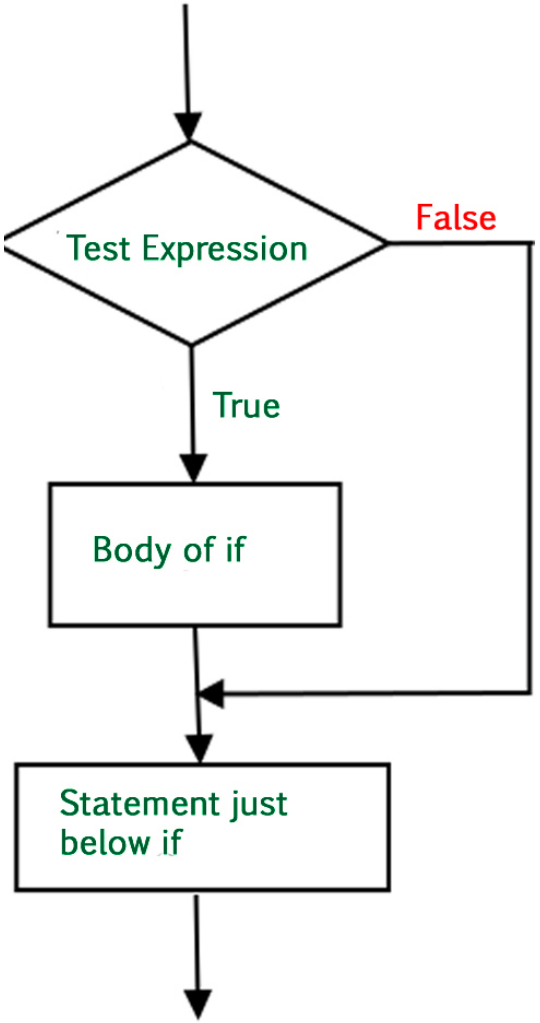
Iterative(Looping): A set of statements are executed repeatedly until the condition becomes false.



Transfer: Statements used to alter the program's way of execution.

4.1 if statement

Boolean value 1.
Comparison Ops:
==,>,>=,<,<=,!=
2. Logical Ops:
and, or, not
3. `bool(num)`:
Non-0→True; 0→False.
4. Identity Ops:
is, is not
5. Membership Ops:
in, not in



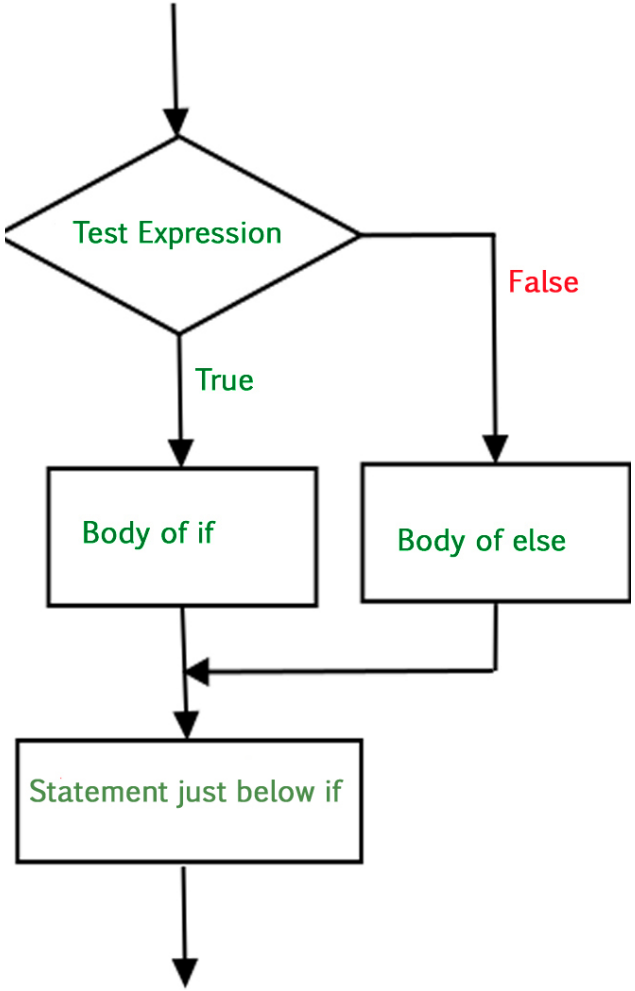

```
if condition :
    # Statements to execute
    # if condition is true
```

Python uses indentation to identify a block.

```
# python program to
      illustrate If statement
i = 10
if (i > 15):
    print ("More than 15.")
print ("Not in if block.")
```

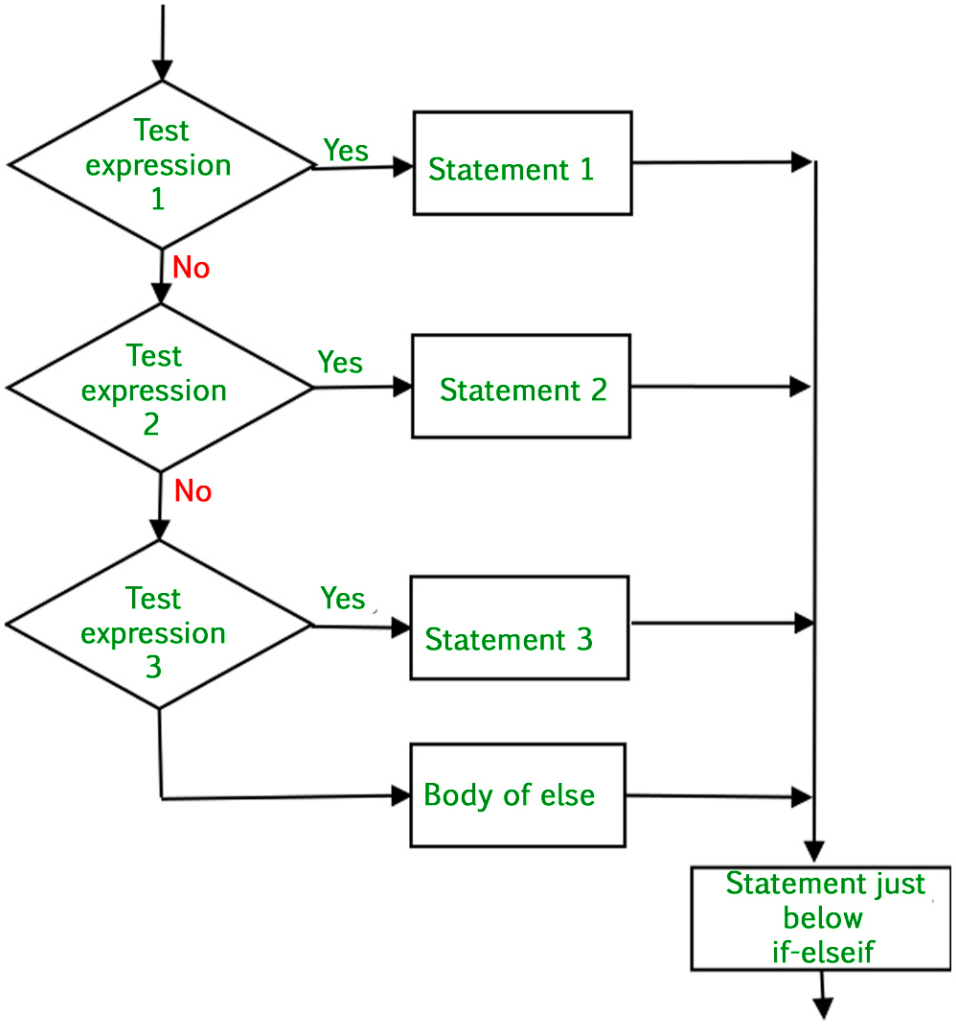
```
x=5
print (x)
print (x==5)
type(x>10 or x<1)
```

if-else



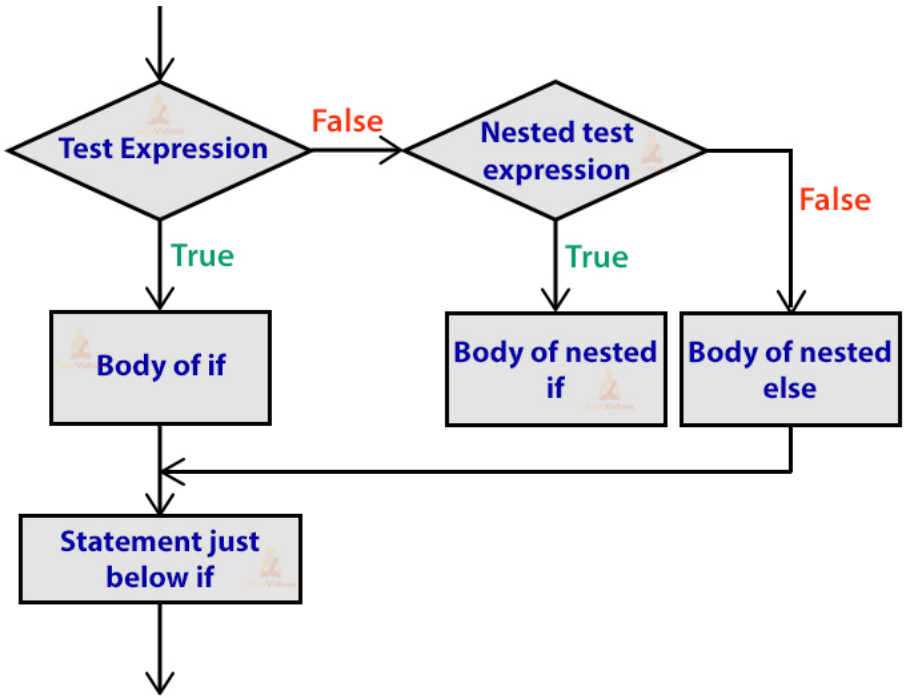

```
if condition :
    # Statements to execute
    # if condition is true
else :
    # Executes this block
    # if condition is false
```

if-elif-else



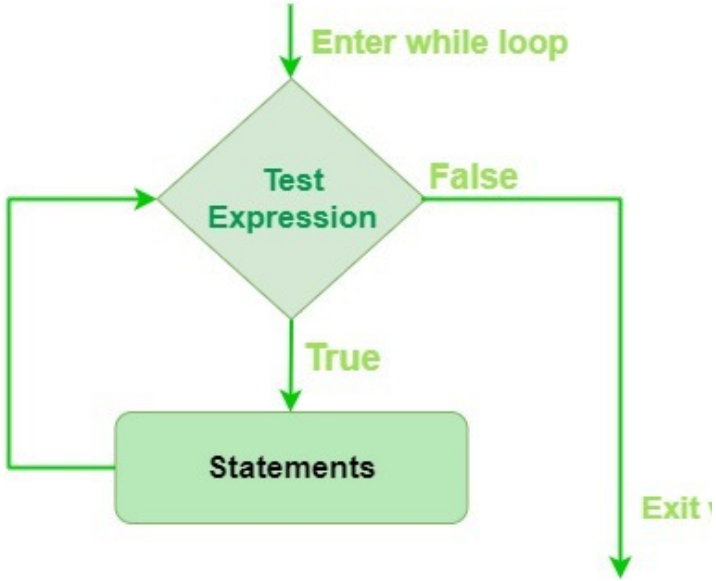

```
# Python if-elif-else ladder
#!/usr/bin/python

i = 20
if (i == 10):
    print("i is 10")
elif (i == 15):
    print("i is 15")
elif (i == 20):
    print("i is 20")
else:
    print("i is not present")
```

Nested-if




```
# python nested If statement
#!/usr/bin/python
i = 10
if (i == 10):
    # First if statement
    if (i < 15):
        print ("smaller than 15")
    # Nested - if statement
    # Will only be executed if
    # statement above it is true
    if (i < 12):
        print ("smaller than 12 too")
    else :
        print ("greater than 15")
```

4.2 while statement




```
# while loop
x = 0
while x < 3:
    x = x + 1
    print("Hello", x, "!")
else:
    print("Alarm: X>=3!")
```

1. Check if the condition given is met (in this case if x<3).
2. If it is, it executes the indented block of code immediately following; if not, it skips the block.
3. The program then loops back from the end of the block to the beginning and checks the condition again.

With else statement: else block will be executed once (and once only) if and when the condition in the while statement fails.

The FIBONACCI SEQUENCE

```
f1 = 1
f2 = 1
while f1<=1000:
    print(f1)
    fnext = f1 + f2
    f1 = f2
    f2 = fnext
print("The end!")
```

```
f1,f2 = 1,1
while f1<=1000:
    print(f1)
    f1,f2 = f2,f1+f2
```

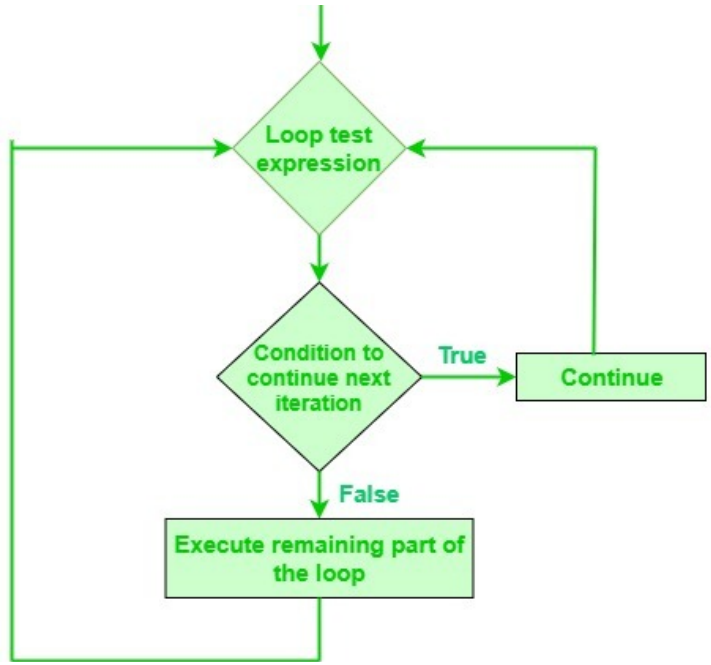
Break and continue

1. break statement: break out of a loop even if the condition for leaving in the while statement is **not met**.

2. continue statement at anywhere in a loop will make the program skip the rest of the **indented code**, and then goes back to the beginning of the loop.

3. **Realization**: Nested **if** controlling inside the loop.

```
Loop{
    Condition:
        break
    }
```



5. List and Array

LIST: **quantities**, one after another, which are called **elements**.

[3, 0, 0, -7, 24], [1, 2.5, 3+4.6j]

1. Individual elements: r[0], r[1], r[2], ...
2. Any type of quantity is allowed.
3. Elements do not have to be the same type.
4. Build-in functions: `max, min, sum, len`
5. Meta-functions 高阶函数:

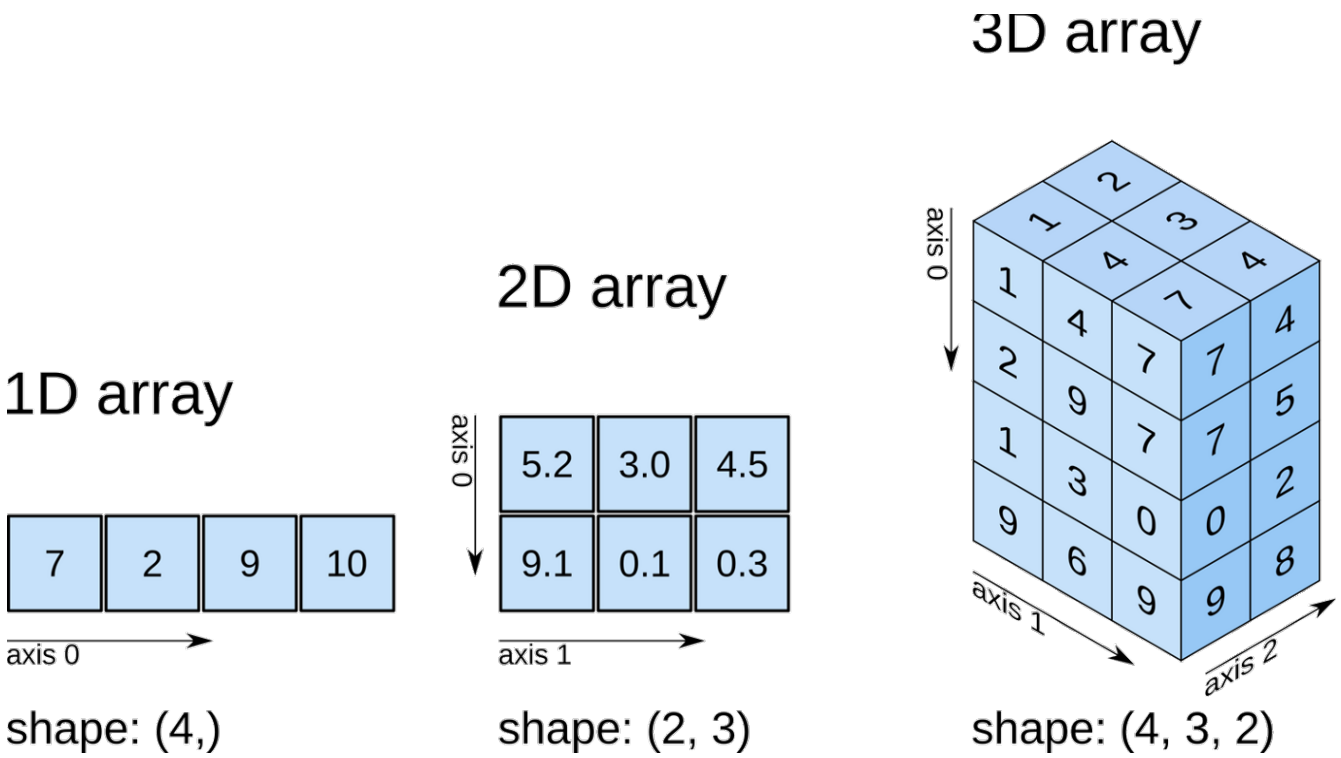
map allows you apply ordinary functions to all the elements of a list at once, and create an *iterator* object.

*.append(), *pop(n)

```
from math import sqrt
r = [ 1, 1, 2, 3, 5, 8, 13, 21 ]
x = 1.0
y = 1.5
z = -2.2
r1 = [ x, y, z ]
r2 = [ 2*x, x+y, z/sqrt(x**2+y**2) ]
print(r1, '\n', r2)

l= sqrt(r[0]**2+r[1]**2+r[2]**2)
print(l)

r[1] = 3.5
print(r)
total = sum(r)
mean = sum(r)/len(r)
print( total ,mean)

logr = list(map(log,r))
r = []
r.append(6.1)
print( logr , r)
```

ARRAY: an ordered collection of elements of the same data type, and fixed number of elements. [3 0 0 -7 24], [1+0j 3+4.6j]

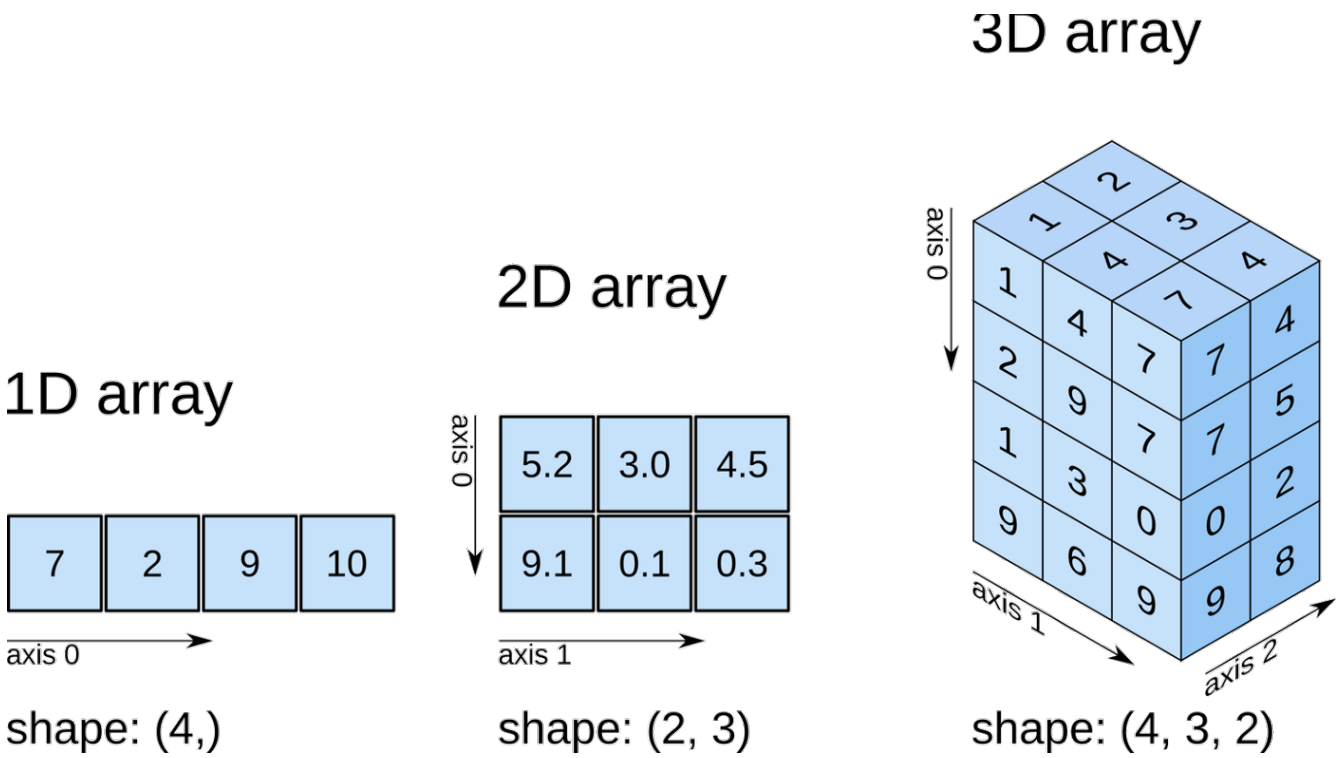
Advantages over lists: **Convenient** for vectors, matrices, high dimensional data. **Versatile and fast** for arithmetic calculations.



1D, just number; 2D, row×column; $\cdots$; the elements are stored **row by row** (nested lists): [5.2,3.0,4.5,9.1,01,0.3]

ARRAY: an ordered collection of elements of the same data type, and fixed number of elements. [3 0 0 -7 24], [1+0j 3+4.6j]

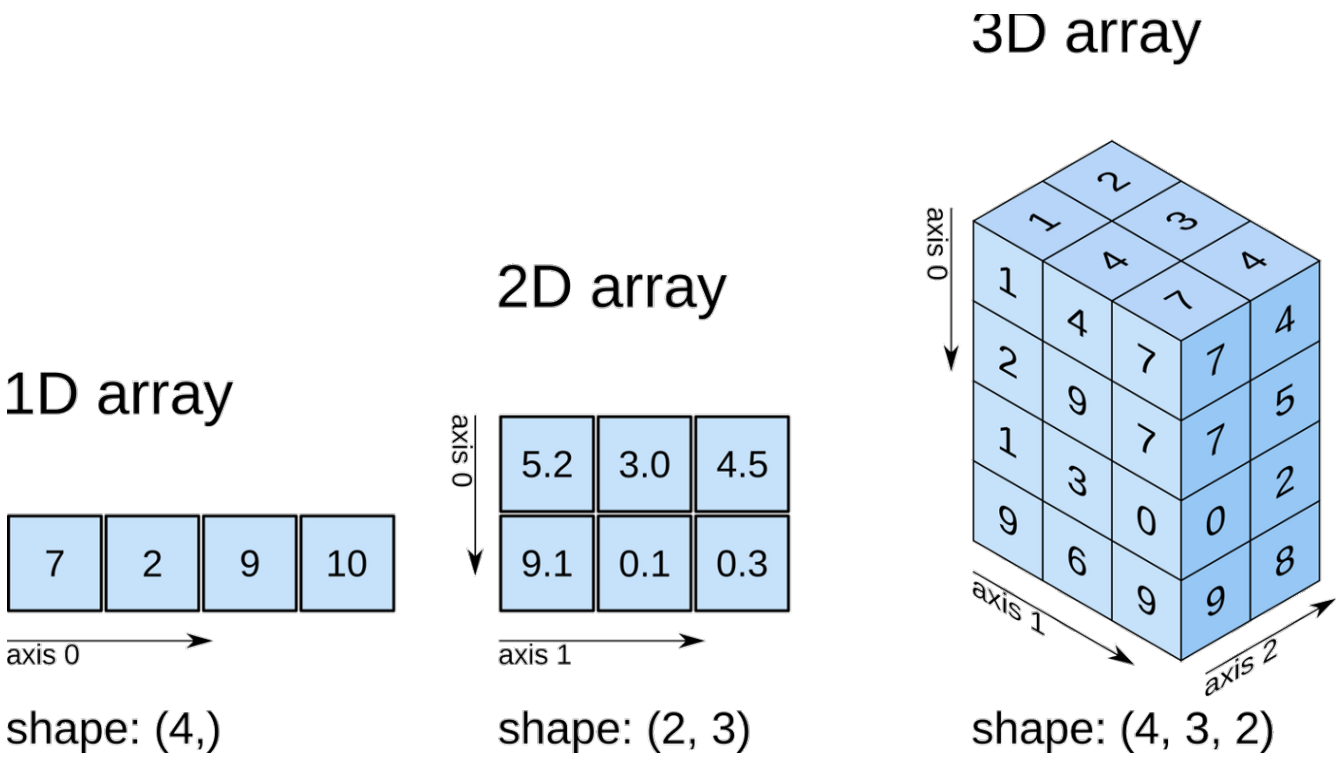
Advantages over lists: **Convenient** for vectors, matrices, high dimensional data. **Versatile and fast** for arithmetic calculations.



1D, just number; 2D, row×column; $\cdots$; the elements are stored **row by row** (nested lists): [5.2,3.0,4.5,9.1,01,0.3]

ARRAY: an ordered collection of elements of the same data type, and fixed number of elements. [3 0 0 -7 24], [1+0j 3+4.6j]

Advantages over lists: **Convenient** for vectors, matrices, high dimensional data. **Versatile and fast** for arithmetic calculations.



1D, just number; 2D, row×column; $\cdots$; the elements are stored **row by row** (nested lists): [5.2,3.0,4.5,9.1,01,0.3]

Create and define an array:

```
import numpy as np
#from numpy import zeros,ones,array

a1 = np.zeros(4, float )
a2 = np.empty(4,float) #let it be!
#two dimension matrix
a3 = np.ones ([3,4], float )

# convert a list into an array
r = [ 1.0, 1.5, -2.2 ]
a4 = np.array( r , float )
a5 = np.array ([[1,2,3],[4,5,6]], int )
a5 [1,2] = 5 # change element value

print (a1, "\n",a2, "\n",r , "\n",a4, "\n",a5)
b1 = a # b1 is a new name of a!
b2 =np.copy(a) # b2 is a copy of a!
print (b1,b2)
```

Read array from a file. e.g., experimental data, calculated results.

values.txt:

```
cat values.txt

1 2 3 4
3 4 5 6
5 6 7 8
```

```
from numpy import loadtxt
a = loadtxt("values.txt", float )
print (a)

[[1. 2. 3. 4.]
 [3. 4. 5. 8.]
 [6. 5. 6. 7.]]
```

Arithmetic with arrays

- Individual elements of an array behave like ordinary variables. a[2]=a[3]+4*a[7]
- Arithmetic with entire arrays at once:
  a) do independently to each element of the array ($a \times 2$);
  b) do between the two elements of the same index ($a+b$, $a*b$, the same shape and size).
- for 1 D array only: len, max, min, map (similar to list)
- for any array: size and shape. e.g, print(a.size, a.shape)
- Array in physics: functions in the numpy package that are useful for performing calculations with arrays. `numpy.dot`

```
from numpy import array,dot
a = array ([1,2,3,4], int)
b = array ([2,4,6,8], int)
print (dot(a,b))
```

```
a = array ([[1,3],[2,4]], int)
b = array ([[4,-2],[-3,1]], int)
c = array ([[1,2],[2,1]], int)
print (dot(a,b)+2*c)
```

Example 2.2 page 62-65

Calculate: 1. the mean; 2. the mean-square (the arithmetic mean of the squares value); 3. the geometric mean(the nth root of the product of n numbers) of a set of numbers stored in a file values.txt.

```
from numpy import loadtxt
values = loadtxt("values.txt", float)
#1 & 2---
mean = sum(values)/len(values)
mean_s = sum(values*values)/len(values)
#3-1---
from numpy import loadtxt
from math import log, exp
logs = array(map(log,values), float)
geometric = exp(sum(logs)/len(logs))
#3-2
from numpy import loadtxt,log
from math import exp
values = loadtxt("values.txt", float)
geometric = exp(sum(log(values))/len(values))
```

The geometric mean:

$$\bar{x} = \left[\prod_{i=1}^{n} x_i\right]^{1/n}$$

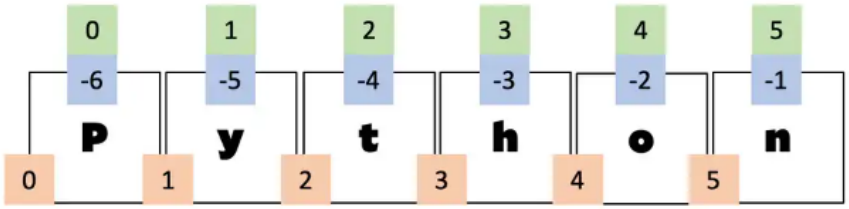
Taking natural logs:

$$\ln \bar{x} = \frac{1}{n}\sum_{i=1}^{n} \ln x_i$$

$$\bar{x} = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \ln x_i\right)$$

Slicing of arrays and lists

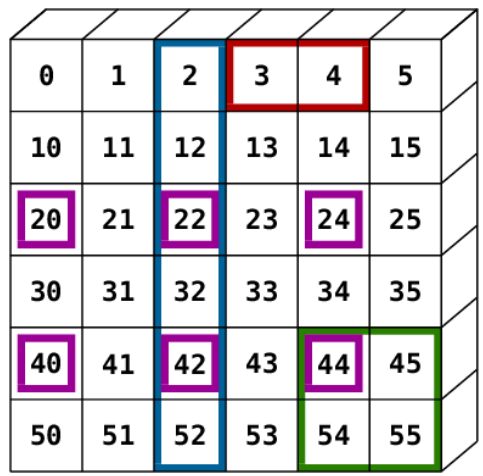
r is a list/array, its sub-list $r[m:n]$ is another list/array composed of elements of r starting with element m and going up to but not including element n.




```
li = list("Python")
start = 0
stop = len(li)
step = 2
slice_obj = slice(start, stop, step)
# an object transfers parameters for
    slicing
print(li[start:stop:step] == li[
    slice_object]) # True
print(li[:], li[1:], li[:5], li[0:-1:3])
print(li[1:6] # ? %when stop value is
    higher than the length of the list,
    it will return the whole list.
```

2D array:

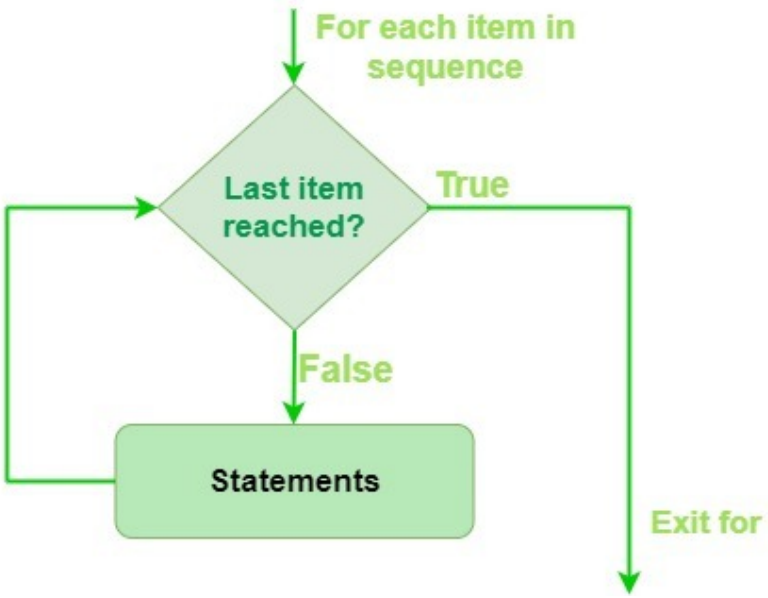
```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



6. “ For ” loops

A for loop runs through the elements of a list or array in turn.

For loop is used when the number of iterations is known.



`range(start,stop,step)`: generate integer values (*iterate*) run from "start" up to, but not including, the "stop", in steps of "step".

```
range(5) # range(0,5,1)
range(2,6);
range(2,20,3);
range(20,2,−3)
```

```
r = [ 1, 3, 5 ]
for n in r:
    print(n)
    print(2*n)
print("Finished")
```

```
for n in range(1,11):
    print(2**n)
```

```
p = 10
q = 2
for n in range(p//q): #p/q a real number!
    print(n)
```

With numerical python package:

1. `arange(start,stop,step)`: similar to `range()` but generate arrays, rather than lists or iterators, with both floating-point and integer values.

2. `linspace(start,stop,num)`: generate a floating point 1D array of the size of "num", which divides the interval from "start" to "stop" (including).

```
import numpy as np
np.arange(1,8,2)  #[1,3,5,7]
np.arange (1.0,8.0,2.0)  # [1.0,3.,5,7]
np.arange (2.0,2.8,0.2)  # [2.0,2.2,2.4,2.6]
np. linspace (1,8,2)  #[1.0,8.0]
np. linspace (1.0,8.0,3)  # [1.0,4.5,7.0]
np. linspace (2.0,2.8,5)  #  [2.0,2.2,2.4,2.6,2.8]
```

Examples

2.6: performing a sum:

$$s = \sum_{k=1}^{100} (1/k).$$

```
s=0.0
for k in range(1,101)
    s+=1/k
print(s)
```

sum of squares of a set of real values in a file "val.txt".

```
import numpy as nm
values = nm.loadtxt("val.txt", float)
s=0.0
for x in values:
    s += x**2
print(s)
```

Q: sum() or for loop?

2.7: (revisit) spectrum of the hydrogen atoms (use nested for loops).

```
R = 1.097e-2
for m in [1,2,3]:
    print("Series for m =",m)
    for k in [1,2,3,4,5]:
        n = m + k
        invlambda = R*(1/m**2-1/n**2)
        print(" ",1/invlambda," nm")
```

```
R = 1.097e-2
for m in range(1,4):
    print("Series for m =",m)
    for n in range(m+1,m+6):
        invlambda = R*(1/m**2-1/n**2)
        print(" ",1/invlambda," nm")
```

Homework

2.9 Madelung constant.

2.10 Nuclear binding energy.

7. User defined functions

Specialized functions, beyond Python standard functions, to perform particular calculations, are often needed.

Factorials of integers

$$n! = \prod_{k=1}^{n} k.$$

```
f = 1.0
for k in range(1,n+1):
    f *= k
```

```
def factorial (n):
f = 1.0
for k in range(1,n+1):
    f *= k
return f

a= factorial (10)
b= factorial (1000)
print a,b
```

In cylindrical coordinates, the distance d from a point (r, θ, z) to the origin.

```
def distance(r,theta,z):
    x = r*cos(theta) y = r*sin(theta)
    d = sqrt(x**2+y**2+z**2)
    return x,y,z,d

x,y,z,d=distance(r,theta,z)
```

```
def f(x,y):
    # Some calculations here ...
    return a,b
```

- **Arguments and returned values**: can be any type of data.
- **Local variables**: are created inside the definition of a function, exist only inside that function.
- **User defined statements**: functions without any returned value.
- **Library**: *.py file contains functions, be imported as xx.

Homework

2.11 Binomial coefficients (p.82)

2.12 Prime numbers (p.83)

2.13 Recursion (p.83)

8. Good programming style

- Include comments in your programs.
- Use meaningful variable names.
- Use the right types of variables.
- Import functions first.
- Give your constants names.
- Employ user-defined functions, where appropriate.
- Print out partial results and updates throughout your program.
- Lay out your programs clearly.
- Do not make your programs unnecessarily complicated.

8. Good programming style

- Include comments in your programs.
- Use meaningful variable names.
- Use the right types of variables.
- Import functions first.
- Give your constants names.
- Employ user-defined functions, where appropriate.
- Print out partial results and updates throughout your program.
- Lay out your programs clearly.
- Do not make your programs unnecessarily complicated.

8. Good programming style

- Include comments in your programs.
- Use meaningful variable names.
- Use the right types of variables.
- Import functions first.
- Give your constants names.
- Employ user-defined functions, where appropriate.
- Print out partial results and updates throughout your program.
- Lay out your programs clearly.
- Do not make your programs unnecessarily complicated.

8. Good programming style

- Include comments in your programs.
- Use meaningful variable names.
- Use the right types of variables.
- Import functions first.
- Give your constants names.
- Employ user-defined functions, where appropriate.
- Print out partial results and updates throughout your program.
- Lay out your programs clearly.
- Do not make your programs unnecessarily complicated.

8. Good programming style

- Include comments in your programs.
- Use meaningful variable names.
- Use the right types of variables.
- Import functions first.
- Give your constants names.
- Employ user-defined functions, where appropriate.
- Print out partial results and updates throughout your program.
- Lay out your programs clearly.
- Do not make your programs unnecessarily complicated.

8. Good programming style

- Include comments in your programs.
- Use meaningful variable names.
- Use the right types of variables.
- Import functions first.
- Give your constants names.
- Employ user-defined functions, where appropriate.
- Print out partial results and updates throughout your program.
- Lay out your programs clearly.
- Do not make your programs unnecessarily complicated.

8. Good programming style

- Include comments in your programs.
- Use meaningful variable names.
- Use the right types of variables.
- Import functions first.
- Give your constants names.
- Employ user-defined functions, where appropriate.
- Print out partial results and updates throughout your program.
- Lay out your programs clearly.
- Do not make your programs unnecessarily complicated.

8. Good programming style

- Include comments in your programs.
- Use meaningful variable names.
- Use the right types of variables.
- Import functions first.
- Give your constants names.
- Employ user-defined functions, where appropriate.
- Print out partial results and updates throughout your program.
- Lay out your programs clearly.
- Do not make your programs unnecessarily complicated.

8. Good programming style

- Include comments in your programs.
- Use meaningful variable names.
- Use the right types of variables.
- Import functions first.
- Give your constants names.
- Employ user-defined functions, where appropriate.
- Print out partial results and updates throughout your program.
- Lay out your programs clearly.
- Do not make your programs unnecessarily complicated.

Any question?

Thank you!