Sequence data types: String, List, Tuple

Sequence data types are non-scalar objects, which means they can be subdivided.

They are an ordered collection of items (elements), and each element is referenced by an index

-5	-4	-3	-2	-1
elem1	**elem2**	**elem3**	**elem4**	**elem5**
0	1	2	3	4

```
s = 'Hello World'  # string elements are Unicode characters

L = ['pop',46,[1,4,5]]  # list elements can be of any data type

T = ('pop',46,[1,4,5])  # tuple elements can be of any data type
```

Access one element of a sequence - Indexing (String)


```
seq[index] #access one element referenced by one index
```

We can use positive indexes, which start from 0.
Or we can use negative indexes, which start for -1.

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	e	l	l	o		W	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10

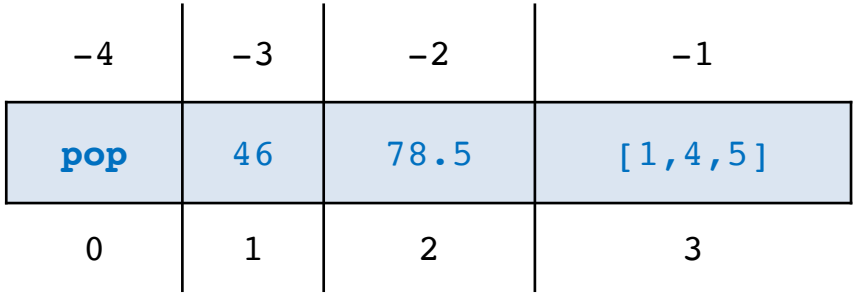
```
s='Hello World'
s[-7]  # 'o'

s[-1]  # 'd'
type(s[-1]) # <class 'str'>

c=s[0]
print(c)
H

s[14] #out of range error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Access one element of a sequence - Indexing (List)

A list is a container of objects

```
seq[index] # access one element referenced by one index
```

```
L = ['pop',46,78.5, [1,4,5]]
```




```
L[1]        # 46
type(L[1]) # <class 'int'>
L[1]*10     # 460

L[0] # pop
type(L[0]) # <class 'str'>
```

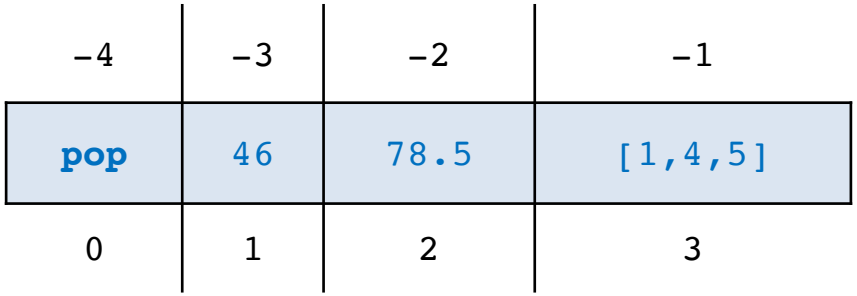
Since the first element is a string type, you can use another [ ] to access an element of that string

```
L[0][0] # p

L[-1]     # [1,4,5]
L[-1][0] # access 1st element of [1,4,5], which is 1
```

Access one element of a sequence - Indexing(tuple)

A tuple is a container of objects

```
seq[ index ] # access one element referenced by one index
```




```
T = ('pop',46,78.5, [1,4,5])
```

```
T[1]      # 46
T[0]      # pop
```

```
T[-1]       # [1,4,5]
T[-1][-1]   # access last element of [1,4,5], which is 5
```

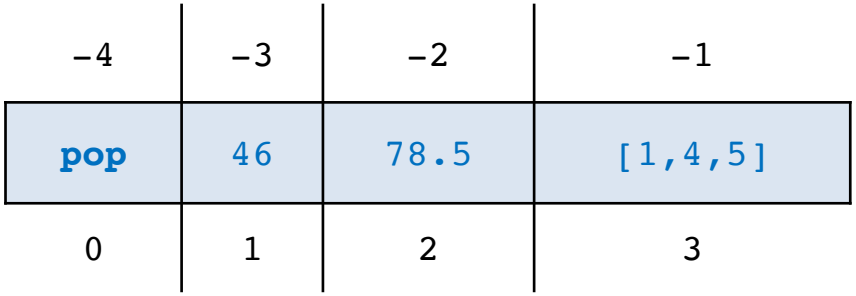
Subsequences can be created with the slice notation


```
seq[start:end]       # from index start through index end-1
seq[start:]          # from index start through last index
seq[:end]            # from index 0 through index end-1
seq[:]               # the whole sequence
seq[::-1]            # reverse the sequence
seq[start:end:step] # from index start through not past end, by step
seq[-2:]             # last two elements in the sequence
seq[:-2]             # everything except the last two elements
```

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	**e**	**l**	**l**	**o**		**W**	**o**	**r**	**l**	**d**
0	1	2	3	4	5	6	7	8	9	10

```
s='Hello World'  # string type
s[1:4]   # from index 1 to index 3 'ell'
s[:4]    # from index 0 to index 3 'Hell'
s[6:]    # from index 6 to the end 'World'
s[1:11:2] # from index 1 to 10 in step of 2 'el ol'
s1=s[:]   # s and s1 reference the same string object (check id)

s2=s[::-1] # reverse the string
print(s2)
```

Subsequences can be created with the slice syntax




```
L = ['pop',46,78.5, [1,4,5]]  # list type

L[1:]   # element at index 1 through last index [46, 78.5, [1, 4, 5]]
L[:3]   # 1st element through element at index-1 ['pop', 46, 78.5]
L[::2]  # from index 0 to last index in step of 2 ['pop', [1, 4, 5]]
L[::-1] # reverse the list

L1=L[:]  # copy of the list - L1 references the copy (check the id)
```

Try to use slicing with a tuple type

```
T = ('pop',46,78.5, [1,4,5])
```

Creating a string object

To create a string, you enclose characters between single quotes or double quotes.

```
s1='Hello World'
type(s1)
```

```
s1="Hello World"
print(type(s1))
```

```
s1='' #we can also define an empty string – the null character
```

```
s1='678656'  #numbers enclosed in quotes are string objects
type(s1)
```

Creating a string object

Escape is used to remove the special meaning of single and double quotes

```
print("\"Hello Python\"")
```

but

```
print(" ' Hello Python ' ")
```

Triple quotes

```
"""
Or you can place COMMENTS inside triple quotes if you
have more than one line of text.
"""

greetings="""Hello word
I wanted to ask you if you are
happy """
```

Creating a string object from a number

the str() function

The str() function is used to convert an int number or float number to a string type

```
str(4) #return a string object
'4'

str(3.141592) #return a string object
'3.141592'
```

Create a string object - The join() method

The join() method is a string method that returns a string by joining all the elements of a container by a join separator.
The join method works with a list of strings or a tuple of strings.

```
'sep'.join(container of strings)
```

```
L=['one','two','three','four'] # homogeneous list of strings

s1=''.join(L1)
print(s1)

s2=' '.join(L)
print(s2)
```

If the container contains any non-string values, it raises a TypeError.

```
T=(1,2,3,4,5)
s3=' '.join(T)
TypeError: sequence item 0: expected str instance, int found
```

Creating a list object

A list type is a container of an ordered sequence of objects. The objects are items of the list. You can define a list with an expression of the form:

```
list_name = [elem1, elem2, elem3, elem4]
```

```
list1 = [1,3,5,7,9]   #homogeneous list, items are all numeric


list2 = ['red','blue','yellow'] #homogeneous list, items are all
string


list3 = ['pop',46,[1,4,5]] #non-homogeneous list, items are of
different types

list4 = []      # empty list
```

Creating a list object - list function

The list function is used to convert different types (like a string, or tuple) to a list type.
The list function creates a list object

```
list((1,2,3,4))  #tuple to list
[1, 2, 3, 4]

list('G+T+C')   #string to list
['G', '+', 'T', '+','C']



D={'lion':3,'elephant':10,'tiger':5}
list(D.keys()) #list of keys
['lion', 'elephant', 'tiger']

list(D.values()) #list of values
[3, 10, 5]

list(D.items()) #list of key,value tuple
[('lion', 3), ('elephant', 10), ('tiger', 5)]
```

Create a list from a string - the split() method

The split method is a string method that breaks up a string at the specified separator and returns a list of strings.

```
str.split(sep) #break up a string at the separator (sep)

str.split() #if sep is not specified, whitespace is the separator
```

Examples

```
s1="a b c d e a aa"
s1.split()
['a', 'b', 'c', 'd', 'e', 'a', 'aa']


s2="a:b:c:d:e"
s2.split(":")
['a', 'b', 'c', 'd', 'e']
```

Try the list function – what is the difference between the split method and list function?

```
list(s2)
['a', ':', 'b', ':', 'c', ':', 'd', ':', 'e']
```

Creating a tuple object

A tuple type is a container of an ordered sequence of objects, like a list type.
You can define a tuple with an expression of the form:

```
tuple_name= (elem1, elem2, elem3, elem4)
```

or

```
tuple_name= elem1, elem2, elem3, elem4
```

```
tuple1 = (1,3,5,7,9)     # homogeneous tuple, items are all numeric

tuple2 = ('a','b','c')   # homogeneous tuple, items are all string

tuple3 = ('pop',46,[1,4,5]) #non-homogeneous tuple, items are of
different types

tuple4 = ()                          # empty tuple
```

tuple object: packing and unpacking

When we create a tuple, we normally assign values to it.
This is called "packing" a tuple:

```
T=10,30,20,40
print(T)
(10, 30, 20, 40)
```

But, in Python, we are also allowed to extract the individual values into variables.
This is called "unpacking":

```
a,c,b,d=T
print(a)
10
```

This feature allows multiple variables assignments to be performed in this way

```
a,b="Hello",[1,3,5]
print(a) #Hello
print(b) #[1, 3, 5]
```

type conversion to tuple

You can use the tuple() function to create a tuple from certain objects

```
tuple('Mary') #string to tuple
('M', 'a', 'r', 'y')
```

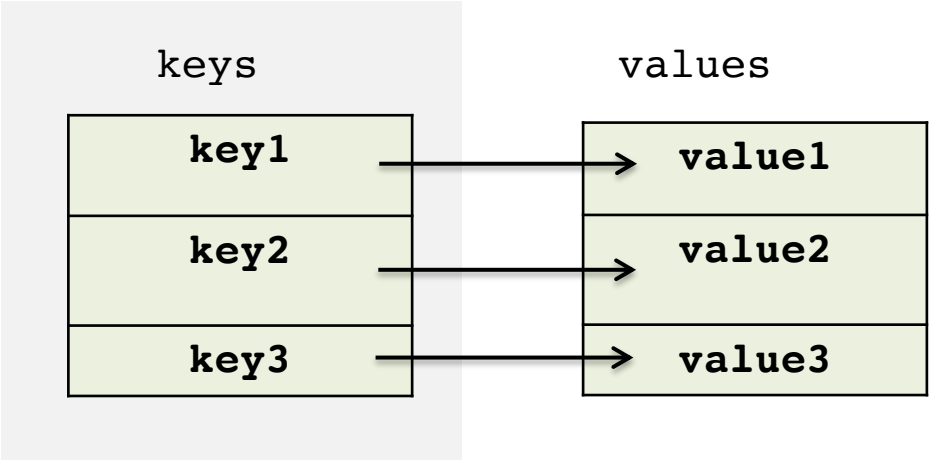
```
tuple([1,2,3,4])   #list to tuple
(1, 2, 3, 4)
```

Dictionaries

Dictionaries contain a collection of items that are pairs of keys and values, and it is a mapping data type.

```
d = {key1 :  value1, key2 : value2, key3 : value3}
```

- Keys can be **only** immutable data types, as strings or numbers
- Values can be of any data types




```
D={'lion': 3, 'elephant': 10, 'tiger': 5}
```

```
empty = {} # empty dictionary
```

Creating a Dictionary: dict() and zip()

Below are several ways of creating this same dictionary

```
D={'lion': 3, 'elephant': 10, 'tiger': 5}
```

You can use the dict() function

```
D1 = dict([('lion', 3), ('elephant', 10), ('tiger', 5)]) #from
a list of 2-elements tuple
```

You can create a dictionary from two lists, with dict(zip())

```
num=[3, 10, 5]
name=['lion', 'elephant', 'tiger']

D2=dict(zip(name,num)) #from two lists
```

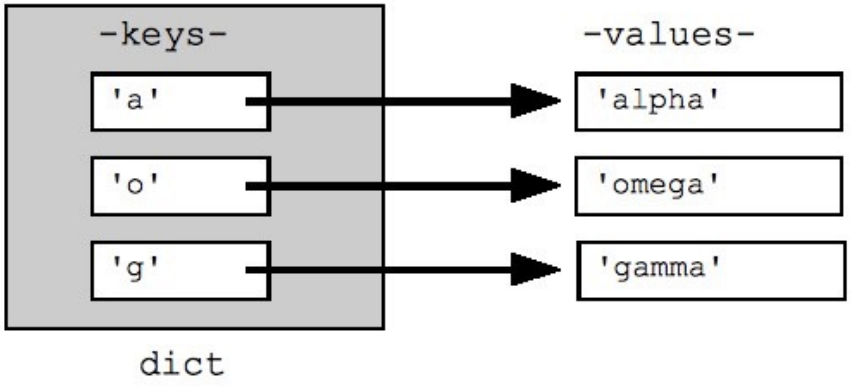
Dictionary - Accessing values

Dictionary is a mapping data type
You can access an individual value in a dictionary by **looking up a key** instead of an index.

```
d[key1] #access the value associated with key key1
```

You can access one individual value - only one key within brackets

```
d={'a':'alpha', 'o':'omega','g':'gamma'}
```




```
d['a']     #access value 'alpha'
d['a'][2]  #access character 'p' of 'alpha'
```

Dictionary does not support indexing, slicing, or other sequence-like behavior

Lists vs Dictionaries

List is a sequence data type

You can access an individual item in a list by its **index**.

```
seq[index]
['lion', 'elephant', 'tiger']
L[2] #tiger
```

Dictionary is a mapping data type

You can access an individual value in a dictionary by **looking up a key** instead of an index.

```
d[key1] #access the value associated with key key1

D={'lion': 3, 'elephant': 10, 'tiger': 5}
D["tiger"] #5
```

Dictionary can store more info, for example how many animals we have

The sys module and the argv attribute

`argv` is an attribute of the sys module used to pass a list of command line arguments to a Python script.
The argv attribute is similar to the $@ in bash.

Make a script called argv_test.py

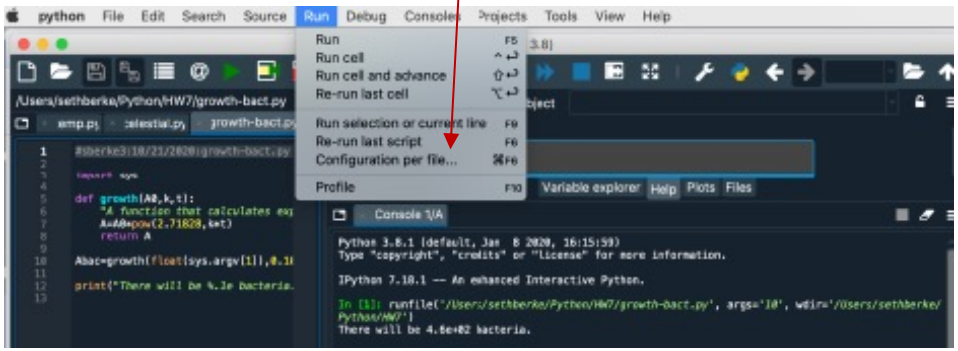
You can open a bash terminal and run the python script like this
`python3 argv_test.py test1 test2 test3 100 200`

```
from sys import argv
print(argv)      #argv is a list of command line arguments
['argv_test.py', 'test1', 'test2', 'test3', '100', '200']

print(argv[0])  #argv[0] is the name of your script
print(argv[1])  #argv[1] is the first argument
print(argv[1:])

#keep in mind that argv is a list of strings
```

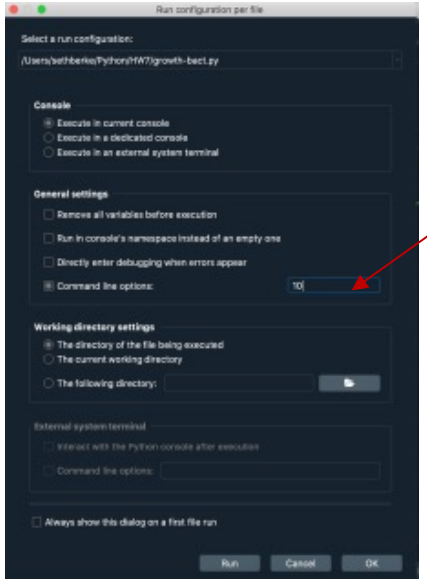
sys argv and editors

To pass arguments to a script in Spyder do the following:

1. Click Run → Configuration per files



2. Enter all the arguments in command line options




```
test1 test2 test3 100 200
```

3. Run the code

sys argv and editors

Passing arguments from IPython Console:

You can also pass arguments by running the script at IPython condole in Spyder like this:

```
runfile('path to the file', args='arg1 arg2 arg3')
```

You can use the absolute path

```
runfile('/Users/maria/argv_test.py', args='test1 test2 test3 100 200')
```

or a relative path, relative to your current directory in IPython – use pwd if you want to know your current directory

```
runfile('argv_test.py', args='test1 test2 test3 100 200')
```

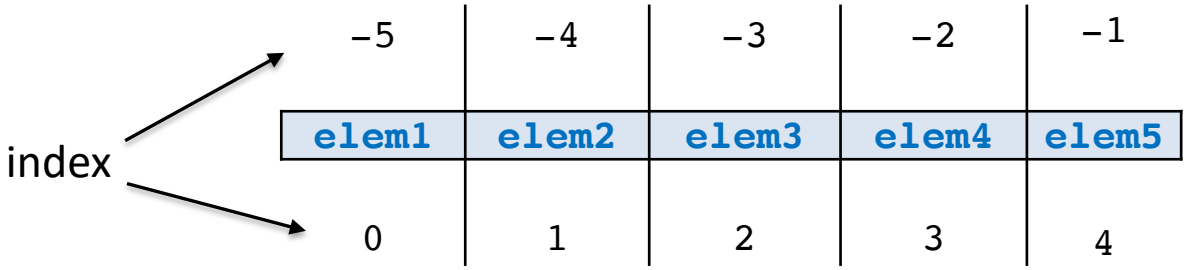
The arguments should be separated by a space, and all enclosed within single quotes.

Summary - Indexing a sequence type: string, list, tuple

Single indexing	
seq[i]	Return item at index i

Slicing	
seq[i:j]	Slice of seq from index i to index j-1
seq[i:j:k]	Slice of seq from index i to index j-1 with step k

```
seq[:]      # the whole sequence
seq[::-1]   # reverse the sequence
seq[-2:]    # last two elements in the sequence
seq[:-2]    # everything except the last two elements
```



Dictionary - Access values

```
d = {key1 :  value1, key2 : value2, key3 : value3}

d[key1] #access the value associated with key key1
```

Data Type Conversion examples

All these functions and methods return a new object which you can store in a variable

`int()`	string to integer	`int('2014')` `2014`
`int()`	floating point to integer	`int(3.141592)` `3`
`float()`	string to float	`float('1.99')` `1.99`
`float()`	integer to float	`float(5)` `5.0`
`str()`	integer to string	`str(4)` `'4'`
`str()`	float to string	`str(3.141592)` `'3.141592'`

`sep.join(iterable)`	list of strings to string	`L=['G', 'A', 'T', 'A']`
		`sep=''`
		`s=sep.join(L)`
		`print(s)`
		`'GATA'`

`sep.join(iterable`	tuple of string to string	`T=('1','2', '33'])`
		`sep='*'`
		`s=sep.join(T)`
		`print(s2)`
		`'1*2*33'`

Data Type Conversion examples

`tuple()`	string to tuple	`tuple('Mary')` `('M', 'a', 'r', 'y')`
`tuple()`	list to tuple	`tuple([1,2,3,4])` `(1, 2, 3, 4)`
`list()`	tuple to list	`list((1,2,3,4))` `[1, 2, 3, 4]`
`list()`	string to list	`list('G+T+C')` `['G', '+', 'T', '+','C']`

`string.split(sep)`	string to list	`s='G+T+C'`
		`L=s.split('+')`
		`print(L)`
		`['G','T', 'C']`

`list(dict.keys())`	dictionary to list of keys	`D={1: 'a', 2: 'b', 3}`
		`list(D.keys())`
		`[1,2,3]`

`list(dict.values())`	dictionary to list of values	`D={1: 'a', 2: 'b', 3: 'c'}`
		`list(D.values())`
		`['a','b','c']`

`list(dict.items())`	dictionary to list of key-value tuple	`D={1: 'a', 2: 'b', 3: 'c'}`
		`list(D.items())`
		`[(1, 'a'), (2, 'b'), (3, 'c')]`

`dict(zip(list1,list2))`	Two lists to dictionary	`l1=[1,2,3]`
		`l2=['a','b','c']`
		`d1=dict(zip(l1,l2))`
		`{1: 'a', 2: 'b', 3: 'c'}`

General syntax

- **Python language is Case Sensitive**

```
bob=100
Bob=90
print(bob)
print(Bob)
```

- **Comment line starts with #**

```
# this is a comment line
```

- **Naming variables**
  - A variable name can start with a letter or underscore
  - All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
  - must not contain spaces or special characters ! @ # % ^ & *
  - must not be any keyword defined in the language, or reserved names: in, for, etc.

```
1v = 2       #error, it starts with a number
v_# = 2      #error, has special character
for="Hello" #error, for is a reserved word

v1 = 2     #variable assignment can have spaces
v1=2
print(v1)
```

Data types

In Python, data are classified in types.

In Python, data take the form of *objects (abstract data types),* and Python programs manipulate these objects

An object's type defines the possible values and operations that type supports.

In this course we will study these built-in types:

- Numeric: Integer, Float
- Sequence: String, List, Tuple
- Mapping: Dictionary

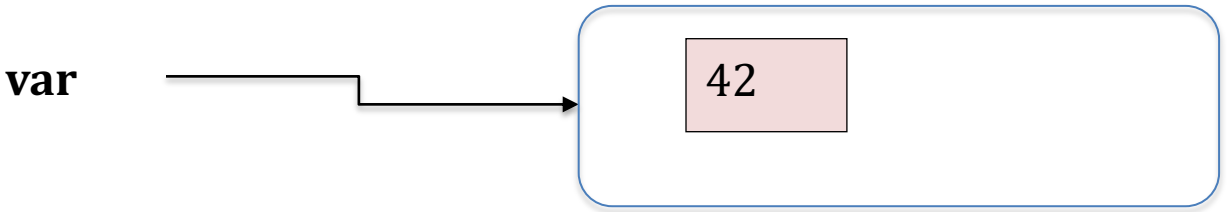
Example	**Data types**
`i = 42`	`int`
`f = 20.5`	`float`
`s = "pop"`	`str`
`L = ["apple", "banana", "cherry"]`	`list`
`T = ("apple", "banana", "cherry")`	`tuple`
`D = {"name" : "John", "age" : 36}`	`dict`

Data: Value, type and identity

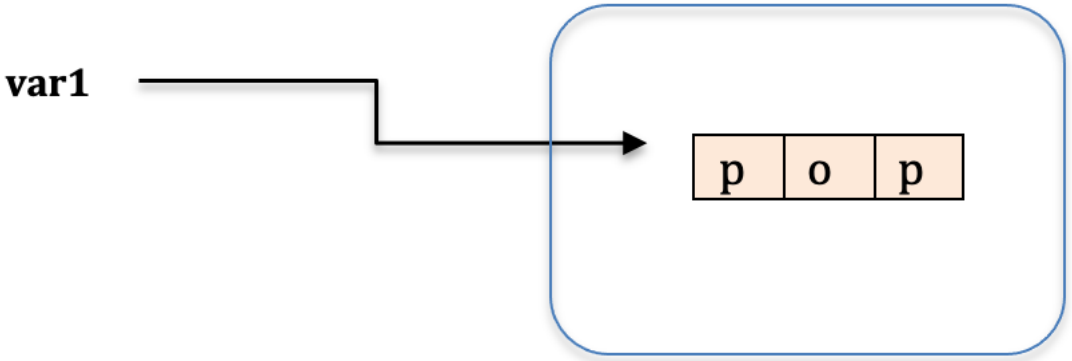
Every object (data type) has these three attributes:

- **Value** stored by the object.
- **Type:** the kind of object that is created. integer, list, string etc. ***type()* function**
- **Identity:** the address that the object refers to in the computer's memory. ***id()* function**

```
var = 42
print(var)
type(var)
id(var)
```




```
var1='pop'
print(var1)
type(var1)
id(var1)
```

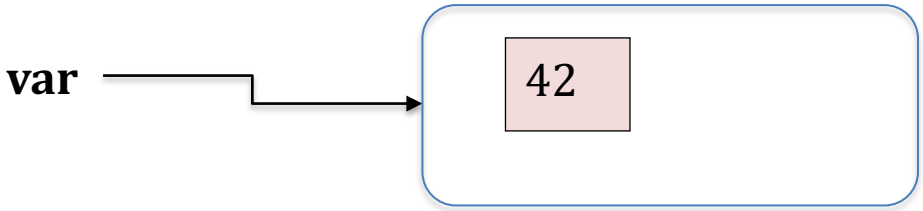


Variables and Object References

What happens when you make a variable assignment in Python?

```
var = 42 #integer object is created and is bound to the name var
type(var)
<class 'int'>
```

An assignment binds the name of a variable to the value stored in the computer memory



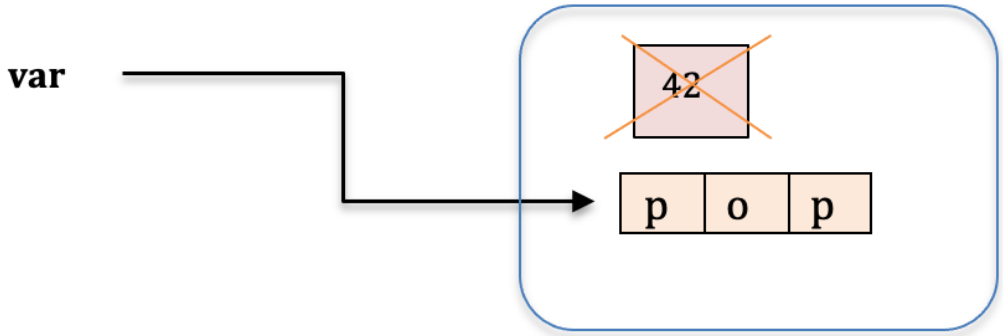
You can check the memory address of a variable by using the function *id( )*

```
id(var)
140266786844240
```

Variables and Object References

Now we change the value of variable var

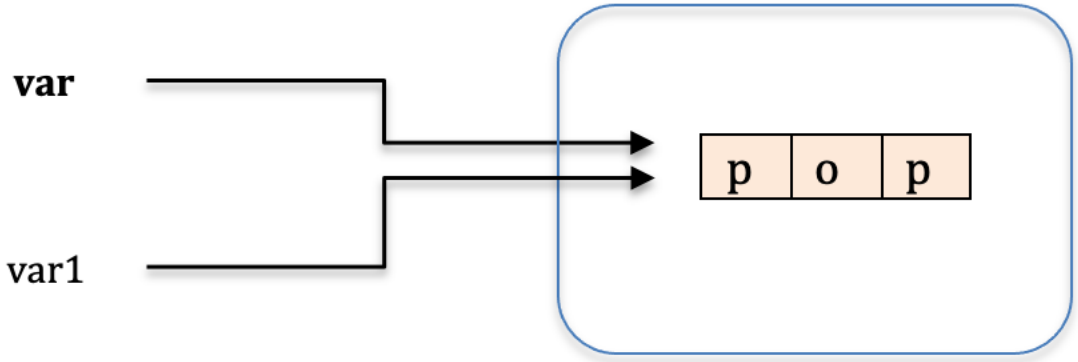
```
var = "pop"  #string object is created and is bound to the name var
type(var)
<class 'str'>

id(var)
140266786848496
#compare with previous id
```



- **Python variables are references to objects, but the actual data is contained in the objects.**
- Python uses dynamic binding for variables.
- In Python, the data type of a variable can change during program execution.

Variables and Object References


```
var="pop"
var1 = var
id(var)
id(var1)
#id is the same
```



Both variables reference the same object.

In this case, Python will not make a copy of the object (value) but will make a new reference to the same value.

This is important to know when we will talk about mutable objects!

Multiple Assignment

Can assign a single value to several variables simultaneously

```
a = b = c = 1
```

Can assign different values to different variables in one line. This is called tuple unpacking.

```
a,b,c = 1,"john",[1,2,3,4]
```

Scalar objects: Numeric type

Scalar objects - cannot be subdivided.

- int represents integer, ex 5
- float represents real number, ex 3.45
- Bool represents Boolean values True and False
- NoneType has only one value None – it represents the absence of a type

In Python, numeric data type represents the data which has numeric value

```
var=42
print(var)
print(type(var))
```


```
var1=3.14
print(var1)
print(type(var1))
```

Operations with Numeric types

Expressions - You can combine objects and operators to form expressions.

`<object> <operator> <object>` → `expression evaluates to a value`

```
i=8
j=2
type(i+j)

f=1.2
type(f*j)
```

Object Operator Object	result
int + int int – int int * int int**int	int
int/int	float
int + float int – float int * float int/float int**float float**int	float
float + float float – float float * float float/float float**float	float

```
z=i/j      #result is stored in variable z
type(z)
i % j      #the reminder of a division
```

Operations with Numeric types

Operators and order of increasing precedence:

Left → Right flow

`+`	**addition**
`-`	**subtraction**
`*`	**multiplication**
`/`	**division**
`**`	**power**
`()`	**parentheses**

Typical precedence you might expect in math

```
3*2**3      # order of precedence results in 24

(3*2)**3    # order of precedence results in 216
```

number syntax and scientific notation

Try these

```
m1=100,100,100
m2=100100100
```

What is the data type of m1 and m2?

Scientific notation is **a way of writing very large or very small numbers**.
Example: 650000000.0 in scientific notation is 6.5×10^8

```
6.5*10**8   #650000000.0
6.5*pow(10,8) #650000000.0
```

math functions - Here some built-in math functions

```
abs(-4)      #absolute value
pow(2,3)     #power
```

The math module

If you need to perform more advanced operations, such as exponential, logarithmic, trigonometric, or power functions, you can import and use the **math module**

```
help("math") #at the IPython Console to enter documentation page of a module
https://docs.python.org/3/library/math.html #online documentation
```

Generic import. If you want to import all functions and constants available in the math module

```
import math
math.log(100)    #specify the module name followed by the function name
math.pi          #specify the module name followed by the constant name

import math as m  #you can rename the module name in your code.
m.log(x)
m.pi
```

Function import: If you want to only import specific functions and/or constants

```
from math import log, sqrt, pi
log(100)          #specify only the function name
pi                #specify only the constant name
```

Type conversion: int() and float()

Sometimes it is necessary to convert values from one type to another.

For example, you cannot sum a string with an integer:

```
s='1210'
i=1340
print(i+s)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

TypeError message reported for *unsupported operand type(s)* because we wanted to add integer data type to a string data type.

We will use the int() function to convert a string type to a numeric type, and then add the values.

```
s1=int(s)
print(i+s1)

print(i+int(s))
```

Type conversion: int() and float()

Here some examples on int() and float() functions

```
int('2014')  #create integer object 2014 from string  '2014'
2014

int(3.141592) #create integer object 3 from float 3.141592
3

float('1.99') #create float object 1.99 from string  '1.99'
1.99

float(5)      #create float object 5.0 from integer 5
5.0
```

print function


```
print(arg1,arg2,..,argN)
```

- Each argument is a data type.
- The print function will make one space between output arguments.
- Arithmetic operations can be performed within the print function.

```
x=2
y=5
D={1:10,2:20}
L=[1,3,6]
s="Hello"

print(x, y, x/y, x**y)
2 5 0.4 32

print("Hello", x+y, "times")  #text is a string type
Hello 7 times

print(D,L,x**y,s)
{1: 10, 2: 20} [1, 3, 6] 32 Hello
```

Tab and newline characters in print function

```
"\n"   #newline character is a string type
"\t"   #tab character is a string type
```

Try them in a print statement:

```
print("Hello", x+y, "\ntimes")
Hello 100
times

print("Hello", x+y, "\n","times")
Hello 100
 times


print("Hello\ntimes\ttimes")
Hello
times times

print("Hello","\n","times","\t","times")
Hello
 times  times
```

print function and formatting operator %

To specify the format, follow the same syntax as printf() in awk and bash – the difference is the syntax of listing arguments

```
print("format1"  %arg1)
print('format1 format2'  %(arg1,arg2))

%[width].[precision]type

%type
%s  string
%d  integer
%e  scientific notation
%f  real number

print('%.1f' %12.345)
print('%10.2f %.1e' %(12.345, 100000.0))

x=5
y=15.253
print('%.2f' %y)
print('%d %.2f' %(x, y/x))
print("Hi, my name is %.5s and I have %d brother." %("MariaG", 1))
```

Input Function

Input function reads a line from standard input, converts it to a string, and returns that string.
Input function is used to make interactive scripts

Make another script called **my-input.py** and in it use the input function to ask the user to enter a number

```
number=input("Enter a number: ")
```

- print what's stored in the variable number
- print the data type of the variable number
- use the variable number to multiply number by 2
- print the result of the multiplication to screen

Run the script
Did you perform arithmetic multiplication or something else?

Now modify the script in order to output the arithmetic operation

Input Function

Input function reads a line from standard input, converts it to a string, and returns that string.
Input function is used to make interactive scripts.

Make another script called **my-input.py** and in it use the input function to ask the user to enter a two numbers separated by a comma

```
number=input("Enter two numbers separated by comma: ")
```

- print what's stored in the variable number
- print the data type of the variable number
- use type conversion to sum the two numbers

Concatenation and Repetition of sequence types

Expressions for sequential types

```
seq1 + seq2   concatenation of two sequence types
seq1 * n      repetition of the sequence n times
```

```
s1="music"
s2="jazz"
s1+s2       #string concatenation 'musicjazz'
s1+' '+s2   #'music jazz'
s1*3        #string repetition 'musicmusicmusic'
```

```
L1=[1,2,3,4]
L2=[5,6,7,8]
L1+L2          #list concatenation [1, 2, 3, 4, 5, 6, 7, 8]
L1*2           #list repetition [1, 2, 3, 4, 1, 2, 3, 4]
```

```
s1+L # cannot concatenate two different sequential types
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "list") to str
```

Functions of sequence type

len(seq) returns the length (number of elements) of a sequence

min(seq) returns smallest item of seq - works for homogeneous items

max(seq) returns largest item of seq - works for homogeneous items

sum(seq) returns the sum – works for list or tuple of numbers

```
L=[1,2,5,4,100]
T=(3,7.8,10000)
max(L) # 100
min(T) # 3
sum(L) # 112

s='abghtyu'
max(s) # alphabetically 'y'
len(s) # 7
len(L) # 5
```

Methods of sequence types

There are some functions and methods (methods are functions applied by using a dot) that work on a sequence type: string, list, and tuple

`seq.count(elem)` returns the number of occurrences of elem in the sequence

`seq.index(elem)` returns the index of the first occurrence of elem

```
s1="this is a string"
s1.count("i") # 3
s1.index('h') # 1
```

```
L=["cat",'cat','dog','elephant']
L.count("cat") # 2
L.index("cat") # 0
```

Common operations for sequence data types: strings, lists and tuples

seq and seq1 can be a list, a string or a tuple

Operation	Result
seq + seq1	Concatenation of seq and seq1
seq*n	Repetition of seq n times
seq[i]	Return item at index i
seq[i:j]	Slice of seq from i to j-1
seq[i:j:k]	Slice of seq from i to j-1 with step k
len(seq)	Total number of elements in seq
seq.count(x)	Number of occurrences of x in seq
seq.index(x)	Return index of x in seq
min(seq)	Return smallest item of seq
max(seq)	Return largest item of seq
sum(seq)	Return the sum – works on list or tuple of numbers

Mutability

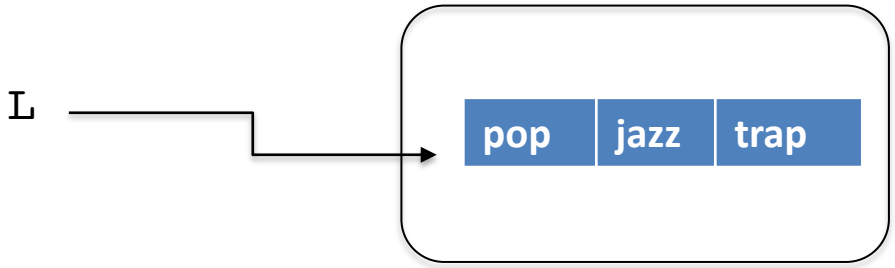
In Python, every object (data type) has these three attributes:

- **Identity:** the address that the object refers to in the computer's memory. *id()* function
- **Type:** the kind of object that is created. integer, list, string etc. *type()* function
- **Value** stored by the object. For example – List=[1,2,3] would hold the numbers 1,2 and 3.

Objects whose values can change are said to be **mutable**

Objects whose values are unchangeable once they are created are called **immutable**.

Lists are mutable Objects



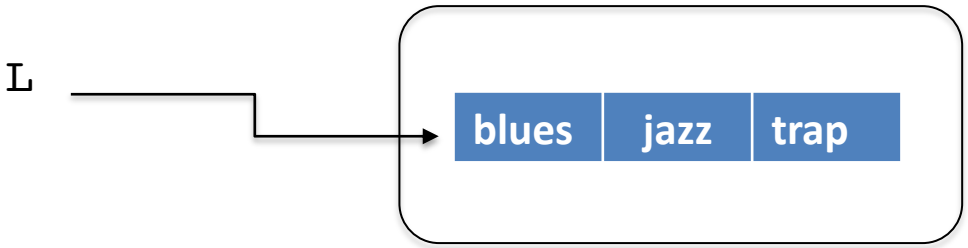

```
L=['pop','jazz','trap']
print(id(L))
print(type(L))
print(L) #values are the elements: 'pop' 'jazz' 'trap'
```

List - Mutability and Methods

Objects whose value can change are said to be **mutable**

```
object[index]=new_value  #item assignment
```

```
L=['pop','jazz','trap']
id(L) #140265716380096
L[0]='blues'
id(L) #140265716380096
```




```
list.append(item) #append method adds an item to the end
```

```
L.append('classic') #returns None
id(L)) #140265716380096
```

List Methods

These list methods return the default None.

```
list.reverse()        Reverse the elements of the list in place.

list.append(item)     Add an item to the end of the list.

list.insert(i,item)   Insert item at index i.

list.remove(item)     Remove the first occurrence of item

list.extend(list1)    Extend the list by joining list1
```

but let's see the pop method

```
list.pop(i)       Remove the item at index i and return that item.
list.pop()        Remove and return the last item in the list.
```

Get info on list methods
https://docs.python.org/3.1/tutorial/datastructures.html

List Method examples


```
Lzoo = ["pangolin", "chicken", "lion"]

Lzoo.append("elephant")
print(Lzoo)
['pangolin', 'chicken', 'lion', 'elephant']

Lzoo.insert(1,"cobra")
print(Lzoo)
['pangolin', 'cobra', 'chicken', 'lion', 'elephant']

Lzoo.extend(["dog", "cat"])
print(Lzoo)
['pangolin', 'cobra', 'chicken', 'lion', 'elephant', 'dog', 'cat']


Lzoo.remove("pangolin")
print(Lzoo)
['cobra', 'chicken', 'lion', 'elephant', 'dog', 'cat']

x=Lzoo.pop(1)
print(Lzoo)
print(x)
['cobra', 'lion', 'elephant', 'dog', 'cat']
chicken
```

List - Delete items with del command

The del statement can also be used to remove an item or a range of items

```
del list[index]
del list[start:end]
del list[start:end:step]
```

```
L=['blues', 'soul', 'trap', 'classic', 'rock', 'punk' ]

del L[0]   #delete element at index 0
print(L)
['soul', 'trap', 'classic', 'rock', 'punk' ]


del L[2:]  #delete a range of elements by using slicing
print(L)
['soul', 'trap']
```

Dictionary methods

Just like Lists, **Dictionaries are** mutable . This means they can be changed after they are created.

`dict.pop(key)`	Remove the key-value and return that value
`dict.popitem()`	Remove the last key-value pair and return it as tuple
`dict.update(dict2)`	Update dictionary dict by adding key:value of dict2

```
D={"apple":2,"cherry":4, "fig":10, "banana":3}

t=D.popitem()
t #('banana', 3)
D # {'apple': 2, 'cherry': 4, 'fig': 10}

x=D.pop("apple")
D #{'cherry': 4, 'fig': 10}
x # 2

D.update({"pear":6, "orange":3})
D #{'cherry': 4, 'fig': 10, 'pear': 6, 'orange': 3}
```

Dictionary– add and remove key:value

```
d[key]=value   #add a new key/value pairs
               #assign a new value to an existing key

del d[key1]   #remove the key1 and the associated value
```

```
Dzoo= {"pangolin" : 5, "sloth" : 3, "tiger" : 20}
```

```
Dzoo["snake"]=6  #add key "snake" and associated value 6
{'pangolin': 5, 'sloth': 3, 'tiger': 20, 'snake': 6}

Dzoo["sloth"]=4 #assign a new value to key "sloth"
{'pangolin': 5, 'sloth': 4, 'tiger': 20, 'snake': 6}
```

```
del Dzoo["tiger"] #remove the key:value pair "tiger":20
{'pangolin': 5, 'sloth': 4, 'snake': 6}
```

More Dictionary methods

```
dict.clear()        Remove all the elements from the dictionary
dict.copy()         Return a copy of the dictionary
dict.get(key)       Return the value of the specified key
```

The length len() of a dictionary return the number of key-value pairs

`len(Dzoo)`

Get info on dictionary methods

https://realpython.com/python-dicts/

String type - Immutability and Methods

Objects whose values are unchangeable once they are created are called **immutable**.

```
s='Hello'
```

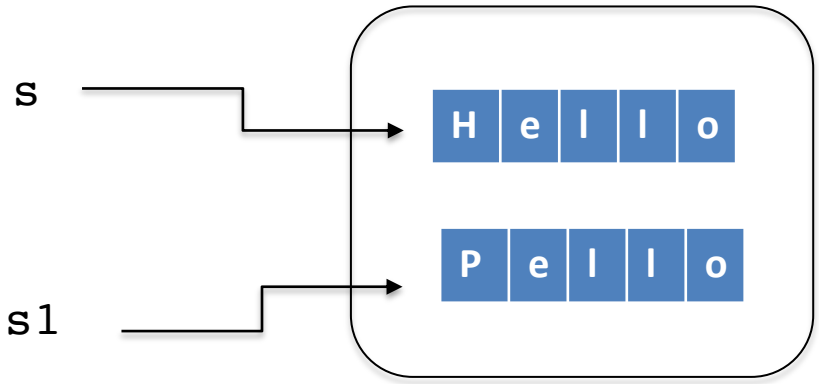
Let's try to change H with P and try to use item assignment.

```
s[0]='P'
TypeError: 'str' object does not support item assignment
```

If you apply a string method, a new string object is created, which you can assign to a new variable.

Let's try to apply the string method replace

```
s1=s.replace('H', 'P')
print(id(s1)
print(id(s))
```



Some string methods

String method examples:

```
s="I like school"


s1=s.upper()  #upper method takes no arguments
print(s1)


s2=s.replace("school","play") #replace takes two arguments
print(s2)
```

More string methods can be found here
https://docs.python.org/3/library/stdtypes.html#string-methods

Combining String Methods and indexing, slicing

You can apply more than one string method on the same string in one command line. Next method applies to the output of the previous method.

```
s1=s.upper().replace("SCHOOL","PLAY")
print(s1)
```

You can slice

```
print(s.upper()[7:])

print(s.upper().replace("SCHOOL","PLAY")[7:])

s2=s.upper().replace("SCHOOL","PLAY")[7:]
print(s2)
```

Tuple type - Immutability

A tuple type support the index and count methods! A tuple is an immutable type

```
T=(42,[1,2,3],'hello')
T[1]="Hello"
TypeError: 'tuple' object does not support item assignment
```

If an element of a tuple is mutable, that element can change. However, the tuple is considered immutable, because the collection of objects it contains cannot be changed.
So, immutability is not strictly the same as having an unchangeable value, it is more subtle.

```
T=(42,[1,2,3],'hello')
id(T) # 140645552536832

T[1].append('pop')
id(T) # 140645552536832

print(T)
(42, [1, 2, 3, 'pop'], 'hello')
```

Python for loop: collection-based iteration

This type of loop iterates over a collection of objects

```
for var in iterable:
    statement(s)     #must indent
```

In Python, an *iterable* object (or simply an *iterable*) is a collection of elements that you can loop (or *iterate*) through one element at a time.
Objects like lists, tuples, dictionaries, and strings are iterable objects.

statement(s) in the loop body are denoted by indentation and are executed once for each item in *iterable*.

the loop variable *var* takes on the value of the next element in *iterable* each time through the loop.

```
L=['foo', 'bar', 'baz']

for i in L:
    print(i)
foo
bar
baz
```

Python for loop: collection-based iteration


```
for var in iterable:
    statement(s)      #must indent
```

Looping over items of a list

```
L=['foo', 'bar', 'baz']

for i in L:
    print(i)
```

Looping over characters of a string

```
s="Monty Python"

for c in s:
    print(c)
```

Looping over keys of a Dictionary

```
dzoo= {"pangolin":5, "sloth":3, "tiger":2, "turtle":10}

for k in dzoo:
    print(k,dzoo[k]) #dzoo[k] is the corresponding value
```

Dictionary methods in for loops - examples


```
dzoo= {"pangolin":5, "sloth":3, "tiger":2, "turtle":10}

dzoo.values() #generate an iterable of values
dict_values([5, 3, 2, 10])

dzoo.items() # generate an iterable of (key,value) pair tuples
dict_items([('pangolin',5), ('sloth',3), ('tiger',2), ('turtle',10)])
```

Looping over an iterable of values by using the values() method

```
for value in dzoo.values():
    print(value)
```

Looping over an itarable of key, value tuples by using the items() method

```
for key,value in dzoo.items():
    print(key,value)
```

Looping over a sequence of integer numbers: range() function

The range() function returns an object of type range, which is an iterable of a sequence of integer numbers

```
range(n)              #from 0 to n-1  and increment by 1
range(start,n)        #from start to n-1
range(start,n,step)   #from start to n-1 with increment specified by step

type(range(5))  #<class 'range'>
range(5)  #generate an iterable of sequence of numbers 0 1 2 3 4
list(range(5)) #convert to list to see the values


for i in range(5):  #0 1 2 3 4
    print(i)

for j in range(2,6): #2 3 4 5
    print(j)

for i in range(2,15,3): #2 5 8 11 14
    print(i)

for i in range(-10,-20,-2): #-10 -12 -14 -16 -18
      print(i)
```

Use of the range function in loops

- You can generate indexes within the loop to access multiple sequence types in one for loop.

```
for index in range(len(sequence):
        print(index,sequence[index])

fruits = ["apple", "banana ", "cherry"]
numbers = [30,15,25]

for i in range(len(fruits)): #loop over generated indexes
    print(i,fruits[i],numbers[i])
```

- We can use the range() function to repeat a set of code a specified number of times

```
for i in range(4):
    some=input("Enter something > ")
    print("Ah .. you entered", some)
```

Using enumerate() in for loops

enumerate(iterable, start=0) takes an iterable and adds a counter to each element, and returns an enumerated iterable object of (count, element) tuples. Count starts from 0 by default.

```
fruits = ["apple", "banana", "cherry"]

type(enumerate(fruits)) # <class 'enumerate'>
enumerate(fruits)       # <enumerate object at 0x7fdcb53747c0>
list(enumerate(fruits)) # convert to list to see the values
[(0,'apple'), (1,'banana'),(2, 'cherry')]
```

fruits

apple	banana	cherry

enumerate(fruits)

0	apple	(0,'apple')
1	banana	(1,'banana')
2	cherry	(2,'cherry')

Using enumerate() in for loops

enumerate(iterable, start=0) takes an iterable and adds a counter to each element, and retuns an enumerated iterable object of (count, element) tuples. Count starts from 0 by default.

You can use enumerate() to generate indices and access values of multiple sequence types in one for loop

```
fruits = ["apple", "banana", "cherry"]
numbers = [30,15,25]


for index, fru in enumerate(fruits):
     print(index, fru, numbers[index])
```

zip() function

zip(iterable1, iterable2, ..., iterableN) takes iterables, aggregates them, and **returns** a zip object, which is an **iterator of tuples**, where the *i*-th tuple contains the *i*-th element from each of the sequences or iterables.

```
fruits = ["apple", "banana", "cherry"]
numbers = [30,15,25]
```

fruits

apple	banana	cherry

numbers

30	15	25

zip(numbers, fruits)

30	apple
15	banana
25	cherry

```
(30,'apple')
(15,'banana')
(25,'cherry')
```

An iterator is an iterable object that can keep track of its location during iteration.

Using zip() in for loops

You can use zip() to iterate over multiple sequences in one for loop (parallel looping)

```
fruits = ["apple", "banana ", "cherry"]
numbers = [30,15,25]

zip(numbers,fruits) #<zip object at 0x7fc6802ef580>
list(zip(numbers,fruits))
[(30, 'apple'), (15, 'banana '), (25, 'cherry')]
```

```
for item1,…,itemN  in zip(iterable1,..,iterableN):
      statements
```

```
for fru,num in zip(fruits,numbers):
      print(fru,num)
```

Using zip() in for loops

zip() provide a **safe way** to handle **iterables of unequal length**, because the iterator stops when the shortest iterable is exhausted, and the elements in longer iterables are left out.

```
fruits = ["apple", "banana ", "cherry"]
numbers = [30, 15, 25, 64, 56, 83]
colors = ["red", "yellow", "pink"]
```

You can use zip() in loops to iterate over multiple sequences of different lengths.

```
for fru, num, col in zip(fruits, numbers, colors):
      print(fru, num, col)
```

Python for loop: summary


```
for var in iterable:
     statement(s)     #must indent
```

for item in list:	Loop over items in a list
for character in string:	Loop over characters in a string
for key in dictionary:	Looping over the keys
for value in dictionary.values():	Looping over the values
for key,value in dictionary.items():	Looping over both keys and values
for num in range(n):	Loop over integer numbers
for index in range(len(sequence)):	Loop over indices of a sequence. Indices can be used to loop over multiple sequences.
for index, item in enumerate(sequence):	Loop over both indices and items of a sequence. Indices can be used to loop over multiple sequences.
for item1,item2 in zip(iterable1,iterable2):	Looping over items of multiple sequences at the same time

Python If-statements


```
#simple if
if test:
    statements
```

If the test is True, the statements get executed
If test is False, nothing happens

```
#if-else
if test1:
    statements1
else:
    statements2
```

If the test1 is True, the statements1 get executed
If test1 is False, the statements2 get executed

```
#if-elif-else
if test1:
   statement1(s)
elif test2:
   statement2(s)
else:
   statement3(s)
```

Once a test is True, the remaining tests are not performed, and it moves to the end

Test condition syntax

Comparison Operators

```
==  equal to
!=  not equal to

>   greater than
<   less than
>=  greater than or equal to
<=  less than or equal to
```

Membership Operators

```
member in container
member not in container
```

Logical operators

```
and
or
not
```

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Test for membership in data types


```
member in container
member not in container
```

Test for membership in strings, lists, tuples, and dictionaries

```
#test if character(s) are in or not in a string
'p' in 'python'
'py' not in 'python'

#test if an item is in or not in a list or tuple
1 in [1,2,3]
1 not in (1,2,3)



D1={1:'a',2:'b',3:'c'}

#test if a key is in or not in a dictionary
1 in D1
1 not in D1

#test if a value is in or not in a dictionary
'a' in D1.values()
'a' not in D1.values()
```

Python If-statements examples


```
#simple if
age = 20
if age > 18:
    print("I can vote") # remember indentation


D1={1:'a',2:'b',3:'c'}
if 'p' in 'python' and 'a' in D1.values():
    print("Yes they are")


D={1:'a',2:'b'}
D1={1:'a',2:'b'}
if D==D1:
    print('They are equal’)



L=[1,2,3]
L1=[1,3,2]
if L!=L1:
    print('They are not equal')
```

Python If-statements examples


```
#if-else
color='red'
guess=input("Guess my color: ")

if color==guess:
    print("You got it")
else:
   print("Sorry")
```

```
#if-elif-else
age = 20
if age > 18:
    print("I can vote")
elif age == 18:
    print("I just turned 18 and can vote too")
else:
    print("I cannot vote")
```

The random module


```
random.random()   #return one random real number in the range [0.0,1.0)
random.randint(a,b) #return one random integer in the range[a,b]
random.choice(seq)  #return one random element from the sequence seq
random.shuffle(L)   #randomly shuffle elements of a list
```

```
import random

random.random()

random.randint(1,6)

L=['green', 'yellow', 'blue', 'orange', 'red']
s='GGCCTTCTCGAATGAATC'
```

The choice() function provides a quick way to randomly select an element from a list or a string:

```
random.choice(s)
random.choice(L)

random.shuffle(L)  #shuffle returns None
print(L)
```

https://docs.python.org/3.8/library/random.html#module-random

The while Loop

```
while test:
        statements
```

Test condition must start off as being True, and then must become false for the while loop to end.

Example:

```
num = 0
while num < 5:
    num = num + 1
    print(num)
```

Can obtain the same with a for loop

```
for i in range(1,6):
      print(i)
```

while loop examples

```
import random
rand_num=0

while rand_num!=8:
      print(rand_num)
      rand_num = random.randint(1,11)
```

We can also use a while True, and if-break

```
while True:
      rand_num = random.randint(1,11)
      if rand_num == 8:
         break
       print(rand_num)
```

while loop examples

```
mynumber=10
number=0   #we set this variable for the while loop to start True.

while number!=mynumber:
      number=int(input("Enter an integer number between 1-10: "))

print("You got the number: " ,number)
```

We can also use a while True, and if-break

```
mynumber=10
while True:
      number=int(input("Enter an integer number between 1-10: "))
      if number==mynumber:
         break

print("You got the number: " ,number)
```

if-break and if-continue statement in loops

The continue statement allows skipping of code within a single loop if criteria have been met. In this case if a match is found go to next entry:

```
for name in ['Newton', 'Galileo', 'Euler']:
    if 'G' in name:
        continue
    print('Hello', name)
```

```
Hello Newton
Hello Euler
```

The break statement will stop the current loop and continue with statements following the loop:

```
for name in ['Newton', 'Galileo', 'Euler']:
    if 'G' in name:
        break
    print('Hello', name)
```

```
Hello Newton
```

Summing loop: sum numbers or construct strings


```
s1 = 0                      # initiate a variable to 0
for i in range(5):          # iterate over iterable
    s1 = s1 + i              # add value and update s1
print(s1)
10
```

The same loop structure can be applied to construct a string, but with the + being the concatenation operator

```
s1 = ''                     # initiate an empty string
for i in range(5):          # iterate over iterable
    s1 = s1 + str(i)         # concatenate value and update s1
print(s1)
# str function is needed in this case, because the values in
range are not strings
01234
```

Nested loops

A nested loop is a loop inside a loop.
The "inner loop" will be executed one time for each iteration of the "outer loop":

```
for element in container:
    for element in container:
        statements(s)
    statements(s)
```

Example:

```
number=[1,2,3]
color=['blue','yellow','red']
```

For every number print each color:

```
for x in number:
    for y in color:
        print(x,y)
```

List Comprehension

List comprehension is an elegant and compact way to make new lists from existing iterables

```
new_list = [expression for member in iterable]
```

The expression is executed for each member in the iterable, and the result will be an item in the new list.

expression: a method, a built-in function, custom function, or any other valid expression that returns a value. That value is an item of the new list

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

fruits_upp = [x.upper() for x in fruits]
print(fruits_upp)
['APPLE', 'BANANA', 'CHERRY', 'KIWI', 'MANGO']


squares = [i*i for i in range(1,11)]
print(squares)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

List Comprehension


```
new_list = [expression for member in iterable if condition]
```

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]
print(newlist)
['apple', 'banana', 'mango']
```

```
Dfruits ={"apple":9,"banana":10,"cherry":3,"kiwi":4,"mango":5}

L=[k for k,v in Dfruits.items() if v > 5]
print(L)
['apple', 'banana']
```

Dictionary Comprehension

Like list comprehensions, Python allows dictionary comprehensions.
We can create dictionaries using simple expressions.
A dictionary comprehension takes the form

```
output_dict = {key:value for item in iterable}
```

```
words = ['data', 'science', 'machine', 'learning']
D={i:len(i) for i in words}
print(D)
{'data': 4, 'science': 7, 'machine': 7, 'learning': 8}
```

```
output_dict = {key:value for item in iterable if condition}
```

```
words = ['data', 'science', 'machine', 'learning']
D={i:len(i) for i in words if len(i) > 5}
print(D)
{'science': 7, 'machine': 7, 'learning': 8}
```

Using zip in dictionary comprehension

Use zip to make a dictionary out of multiple sequences

```
words = ['data', 'science', 'machine', 'learning']
values = [5, 3, 1, 8]

dict_a = {i:j for i, j in zip(words, values)}
print(dict_a)
```

same as

```
dict_a=dict(zip(words, values))
```

```
words = ['data', 'science', 'machine', 'learning']
values = [5, 3, 1, 8]
dict_b = {i:j for i, j in zip(words, values) if j > 4}
print(dict_b)
```

dictionary comprehension

Use items() to make a dictionary out of an exiting dictionary

```
D={'DATA': 25, 'SCIENCE': 9, 'MACHINE': 1, 'LEARNING': 64}

D1 = {i.lower():j*2 for i, j in D.items()}
print(D1)
```

Create lists using Loops

You can use a for loop to create a list of elements in three steps:

1. Initialize an empty list.
2. Loop over an iterable or range of elements.
3. Append (or extend) each element to the end of the list.

If you want to create a list containing the first ten perfect squares

```
squares = []

for i in range(1,11):
      squares.append(i*i)

print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Using list comprehension

```
squares = [i*i for i in range(1,11)]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Create lists using Loops

You can use a while loop to create a list of elements in three steps:

1. Initialize an empty list.
2. make a while loop
3. within the while loop append (or extend) each element to the end of the list.

If you want to create a list of 4 unique integer random numbers in range 1-10

```
import random
L=[]

while len(L)!=4:
      num=random.randint(1,10)
      if num not in L:
         L.append(num)
print(L)
```

Create dictionaries using for Loops

You can use a for loop to create a dictionary in three steps:

1. Initialize an empty dictionary.
2. Loop over an iterable or range of elements.
3. Add key-value pair with D[key]=value or D.update({key:value})

```
veggie=['spinach', 'broccoli', 'edamame', 'bell pepper', 'kale',
'cabbage', 'celery', 'asparagus', 'lettuce']

num=[30,15,25,11,3,4,1,5,6]

D={}
for k,v in zip(veggie,num):
    if 'a' in k and v < 10:
        D[k]=v # or D.update({k:v})

print(D)
{'kale': 3, 'cabbage': 4, 'asparagus': 5}
```

Use dictionary comprehensions

```
D={k:v for k,v in zip(veggie,num) if 'a' in k and v < 10}
print(D)
{'kale': 3, 'cabbage': 4, 'asparagus': 5}
```

Create dictionary using Loops

You can use a while loop to create a list of elements in three steps:

1. Initialize an empty dictionary.
2. make a while loop
3. within the while loop add key-value pair with D[key]=value or D.update({key:value})

If you want to create a dictionary where the keys are 4 integer random numbers in range 1-10, and the values are the corresponding squares

```
import random
D={}

while len(D)!=4:
      num=random.randint(1,10)
      D[num]=num**2

print(D)
```

Create strings using for Loops

You can use a for loop to create a string in three steps:

1. Initialize an empty string.
2. Loop over an iterable or range of elements.
3. Use concatenation to build up the string

```
num=[30,15,25,11,3,4,1,5,6]
```

Make a string whose elements are the numbers in the list

```
s1=''
for i in num:
    s1=s1+str(i) #concatenation
print(s1)
```

```
3015251134156
```

This is a summing loop

```
s1=0
for i in num:
    s1=s1+i  #addition
print(s1)
```

```
100
```

Example:

```
L=[ float(input("Enter a number: "))  for i in range(4) ]
print(L)
```

```
L=[]
for i in range(4):
    num=float(input("Enter a number: "))
    L.append(num)
```

Loops vs comprehensions - optional

- Comprehension is usually faster

```
import time
iterations = 100000000

start = time.time()

mylist = []
for i in range(iterations):
    mylist.append(i+1)

end = time.time()
print(end - start)

start = time.time()
mylist = [i+1 for i in range(iterations)]
end = time.time()
print(end - start)
```

- But pay attention to the size of the list because a list comprehension in Python works by loading the entire output list into memory! You can explore the module **tracemalloc** to trace memory allocations.

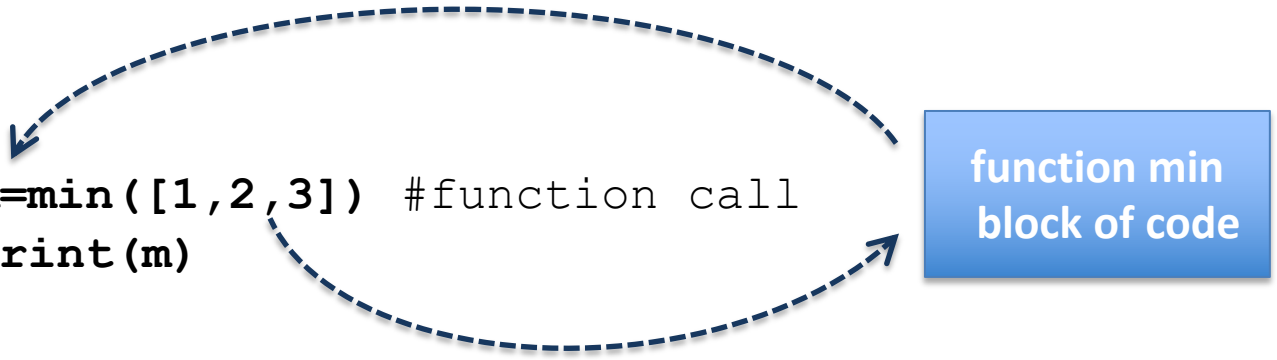
Functions

In programming, a **function** is a self-contained block of code that encapsulates a specific task or related group of tasks.
A function's block of code is executed only when a function is called.

To call a function, use the function name followed by parentheses.

Call built-in functions - you need to know

- What arguments it takes
- What values it returns




```
l=len('Hello')
print(l)
```

Creating your own functions is important

You write some code that implements some task and find that you need that task in many different locations within your script.

Don't Repeat Yourself (DRY) Principle of software development – reducing repetition and avoiding redundancy

Fahrenheit to Celsius formula

$$°C = \frac{°F - 32}{1.8}$$

Math style

$$fc(x) = \frac{x - 32}{1.8}$$

```
xf=100
xc=(xf-32)/1.8
print(xc)
…
…
xf=75
xc=(xf-32)/1.8
print(xc)
```

Code reusability

Define Fahrenheit to Celsius function

```
def fc(x):
    C=(x-32)/1.8
    return C



#Call the function
xc=fc(100)
print(xc)

xc1=fc(75)
print(xc1)
```

Defining and calling functions - return statement

A function's block of code is executed only when a function is called

```
#function definition
def function_name(parameters):
    "function_docstring"
     code block
     return expression

#function call
var=function_name(arguments)
print(var)
```

Make a script called ***myfunc.py:***

```
#Define this function
def fc(x):
     "convert temp from F to C"
     C=(x-32)/1.8
     return C

#Call the function
y=fc(100)
print(y)
y1=fc(75)
print(y1)
```

Return statement

A parameter is a name of a variable listed inside the parentheses in the function definition.
An argument is a value that is assigned to a parameter when a function is called.

```
#Function definition
def fc(x):          # x is a parameter
    "convert temp from F to C"
    C=(x-32)/1.8
    return C



#Function call. Use the function name followed by parentheses
 y=fc(100)        # 100 is the argument

when you call the function
# argument 100 is assigned to the parameter x
  Mechanism of argument passing
# block of code function is executed
# return: the value of C is passed back to the function call
  and can be stored in variable y

 print(y)
```

Defining and calling functions – no return statement

A function's block of code is executed only when a function is called

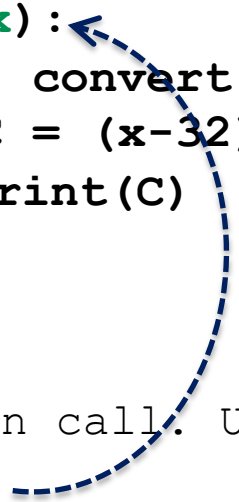
```
#function definition
def function_name(parameters):
    "function_docstring"
     code block

#function call
function_name(arguments)
```

Make a script called ***myfunc1.py***

```
#Define this function
def fc(x):
     "convert temp from F to C"
     C=(x-32)/1.8
     print(C)

#Call the function
fc(100)
fc(75)
```

No return statement


```
#Function definition
def fc(x):          # x is a parameter
        " convert F in C"
        C = (x-32)/1.8
        print(C)



#Function call. Use the function name followed by parentheses
fc(100)             # 100 is the argument
```



```
when you call the function
# argument 100 is assigned to the parameter x
  Mechanism of argument passing
# block of code function is executed
  None is returned
```

Common error: storing None in a variable when calling a function without a return statement

```
x=fc(100)     #in x None is stored
print(x)
None
```

Parameters of a function

A parameter is a variable listed inside the parentheses in the function definition.
An argument is a value that is assigned to a parameter when a function is called.

The number of arguments in the function call should match exactly with the number of parameters in the function definition (positional order).

This function takes **two parameters** and returns one value.

```
def division(par1,par2):    #two parameters
    "divide two numbers"
    D=par1/par2
    return D

# we run the function on two values
y=division(10,5) # we pass two arguments or values
z=division(5,10)
```

Do variables y and z store the same results? Print results to screen. Run the script

Different data types can be arguments of a function


```
def multiply(a,b):
    " multiplication or repetition”
     return a*b


x=multiply(10,20) #the arguments are two numbers
print(x)

y=multiply("hello",4) # the arguments are a string and a number
print(y)
```

Now let’s define the same function but without a return statement.

```
def multiply(a,b):
    " multiplication or repetition”
     print(a*b)

L=[1,2]
n=4
multiply(L,n) # the arguments are a list and a number
```

Try to pass two lists. Would this work?

Practice: function with two parameters and return

Write a script called ***myfunc1.py*** and in it do the following:

a) Make a function called *mysum* that takes two parameters and returns the sum of those two parameters. Write a doc string within the function describing what this function does

b) Call this function on two lists, and store the result in variable **L**. Print the variable

c) Would you be able to use that same function on two strings and two numbers? Answer in a comment line

Function takes no parameters

A function's block of code is executed only when a function is called

This function takes **no parameters,** uses the input function and returns a value

```
def f1():
     num=float(input("enter a number: "))
     print("You entered the number:", num))
     return num*num

y=f1()
print(y)
```

This function takes **no parameters,** uses the input function and does not return

```
def f2():
    num=float(input("enter a number: "))
    print("You entered the number:", num))
    print(num*num)

f2()
```

Practice: Taking input from the input function

Make a script called ***input1.py*** and in it:

a) Make a function *input_and_sum()* that does the following:

- takes no parameters
- uses the input function to ask the user to enter two numbers separated by a comma.
- returns the sum (arithmetic) of those two numbers.

b) Run the function on two numbers, and print the result to screen

Practice: Taking input from the input function, and no return

Make a script called ***input2.py*** and in it:

a) Make a function *input_and_sum()* that does the following:

- takes no parameters
- uses the input function to ask the user to enter two numbers separated by a comma.
- **print** the sum (arithmetic) of those two numbers.

b) Run the function on two numbers. The result should be displayed to screen.
None should not appear in the output.

Functions can return multiple values

Multiple values can be returned as a tuple

```
def f3(arg):
    return arg, arg*2, arg*3

x=f3(10)
print(x)
(10,20,30)

print(type(x)) #tuple type
print(x[0]) #access first element of tuple x - first returned value
```

Common error: you cannot use the return statement more than once in this way:

```
def f3(arg):
    return arg
    return arg*2
    return arg*3
```

Practice: function returns multiple values

Make a script ***myfunc2.py*** and in it:

a. Make a function called *dict_keys_values* that takes any dictionary as an input parameter and returns both the list of keys and the list of values of that dictionary.

b. Run your function on this dictionary and store the result in variable d_out

```
d1 = {'a': 1, 'b': 2}
```

c. Print variable d_out

d. Access values in d_out and print the list of keys

e. Access values in d_out and print the list of values

f. Would you be able to run this function on another data type?
Answer with a comment line

Local and global variables

Local variables can be used only inside the function in which they are declared.

```
def f4(n):
      var=5  #local variable - defined inside a function
      return var*n

print(var)
x=f4(5)
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'myvar1' is not defined
```

Global variables can be used throughout he program, including inside a function:

```
var=5   #global variable - defined outside a function

def f5(n):
     return var*n

print(var)
x=f5(5)
```

Functions can call other functions

We've seen functions that can print text or do simple arithmetic, but functions can be much more powerful than that.
For example, a function can call another function:

```
def square(x):
     s=x*x
     return s

def sum_of_squares(x,y,z):
    sx=square(x)
    sy=square(y)
    sz=square(z)
    return sx+sy+sz

a=1
b=2
c=3
result=sum_of_squares(a,b,c)
print(result)
```

Example: If-statements in functions

If-statements can be in the function's block. For example:

Make a script called if-func.py and define this function:

```
def c1(arg):
    if arg < 5:
        return arg
    else:
        return arg**2

x=c1(3)
print(x)


def c2(arg):
    if arg < 5:
        print(arg)
    else:
        print(arg**2)

c2(3)
```

Example: loops in functions

Loops can be in the function's block . For example:

- In a script called my_upper.py make a function called *string_upper*. The function should take one string as a parameter, convert it to uppercase, and **print** each character on one line. Use a for loop. Run the function an arbitrary string

```
def string_upper(mystr):
    for c in mystr:
        print(c.upper())


string_upper("music")
```

- In a script called **conv_numstring.py,** make a function called *num_str* that takes one integer as parameter, and returns a string of the numbers from 0 to n-1. Example, if the integer value is 6, the returned string will be `'012345'`

```
def num_str(n):
    s1 = ''
    for i in range(n):
        s1 = s1 + str(i)
    return s1


s=num_str(6)
print(s)
```

Example: functions in comprehensions

Functions that have a return statement can be called within comprehensions.

For example:

```
def FCconversion(F):
    """
       Takes a temperature in Fahrenheit F,
       converts it to Celsius C, and return C
       """
    C=(F-32)/1.8
    return(C)
```

We create a list of temperature values in Celsius from temperature values 1-100 in Fahrenheit:

```
L=[ FCconversion(i) for i in range(1,101)]
```

Summary of what we learned about return

Return statement causes your function to exit and hand back a value to its caller

```
def f1(arg):
        return arg*arg

x=f1(10)
print(x)
```

Multiple values are returned as a tuple

```
def f2(arg):
    return arg, arg*2, arg*3

x=f2(10)
print(x)
(10,20,30)
print(x[1]) # print only the second value in the tuple
```

Function with no return - you can not save the result in a variable; in this case the calculated value only gets printed to screen

```
def f3(arg):
    value=arg*arg
    print(value)

f3(10)
```

Summary of what we learned about arguments of a function

This function takes **one parameter**, and returns one value

```
def f4(arg):
      a2=arg*arg
      return a2

x = f4(10) # call the function
print(x)
```

This function takes **two parameters**, and returns one value

```
def f5(arg1,arg2):
      M=arg1*arg2
      return M

y = f5(10,20) # call the function
print(y)
```

This function takes **no parameters**, and returns one value

```
def f6():
      return 'Have a good day!'

z = f6() # call the function
print(z)
```

NumPy Arrays

NumPy (Numerical Python) is a Python library designed for working with numerical data in Python. It is widely used in scientific computing, data analysis, and machine learning.

It provides:

- N-dimensional array object, or ndarray is a fixed-size and homogeneous (fixed-type) multidimensional array. It contains elements of a single data type, such as all integers, all floating-point numbers

- powerful mathematical functions for operating on those arrays of numbers.

- high-performance array calculations , because a ndarray is an homogeneous block of data

In this course we will focus on 1-D and 2-D arrays

NumPy module

Import NumPy in this way

```
import numpy as np    #generic import and rename the module
```

https://numpy.org/doc/stable/user/basics.html

NumPy arrays vs lists

type	items	mutability	size
list	Heterogeneous	mutable	change
numpy. ndarray	Homogeneous	mutable	fixed

```
import numpy as np
L1=[7,2,9,10]


v1=np.array(L1) #converts a list of numbers to a 1D array
print(v1)
[7 2 9 10]

print(type(v1))
<class 'numpy.ndarray'>

print(v1.dtype) #dtype returns the data type of the elements
int64
```

NumPy arrays vs lists

type	items	mutability	size
list	Heterogeneous	mutable	change
numpy. ndarray	Homogeneous	mutable	fixed

All items of a ndarray must be of the same data type or dtype

In this course we will focus on integer (int64), float (float64) , boolean (bool) types

```
L1=[7.2,2,9,10]
v1=np.array(L1)
print(v1)
[ 7.2  2.   9.  10. ]
print(v1.dtype)
float64

L1=[7.2,2,9,10,'pop']
v1=np.array(L1)
print(v1)
['7.2' '2' '9' '10' 'pop']
print(v1.dtype)
<U32  #unicode string
```

NumPy arrays vs lists

type	items	mutability	size
list	Heterogeneous	mutable	change
numpy. ndarray	Homogeneous	mutable	fixed

```
mutability

L1=[7.2,2,9,10]
v1=np.array(L1)

L1[0]=10    #[10, 2, 9, 10]
v1[0]=10    #[10  2  9 10]



L1[:3]=[0,0,0]  #[0, 0, 0, 10]
v1[:3]=0  #[ 0  0  0 10]  #the value is propagated to the
entire selection. Broadcasting
```

NumPy arrays vs lists

type	items	mutability	size
list	Heterogeneous	mutable	change
numpy. ndarray	Homogeneous	mutable	fixed

```
size

id(L1) #140471496928832

L1.append(100) #[0, 0, 0, 10, 100]
id(L1) #140471496928832
#id is the same. Size can dynamically change.



id(v1) #140471497006512

v1=np.append(v1,[100])  #[  0   0   0  10 100]
id(v1) #140471497048784
#id is different. Size is fixed, and a new ndarray object is
created.
```

NumPy arrays vs lists

type	items	mutability	Size	
list	Heterogeneous	mutable	change	
numpy. ndarray	Homogeneous	mutable	fixed	**High-performance array operation**

numeric calculations much easier and faster with ndarray than list

```
L1=list(range(10)) #[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v1=np.arange(10)   #[0 1 2 3 4 5 6 7 8 9]
```

We want to calculate the cube of each element

```
L2=[i**3 for i in L1] #[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
v2=v1**3              #[  0   1   8  27  64 125 216 343 512 729]
```

We want to calculate the root mean square of each element

```
import math
L3=[math.sqrt(i) for i in L1]

v3=np.sqrt(v1) #no need to import math
```

Covert lists/ndarrays

np.array() - lists to ndarrays

```
L1=[7,2,9,10]
v1=np.array(L1) #converts a list of numbers to a 1D array
[ 7  2  9 10]

L2=[[5.2,3,4],[9.1,0.1,0.3]]
M=np.array(L2) #converts a list of lists to a 2D array
[[5.2 3.  4. ]
 [9.1 0.1 0.3]]
```

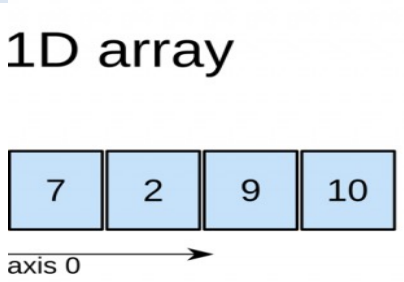
A method is applied to a ndarray object

ndarray.tolist() - ndarrays to lists

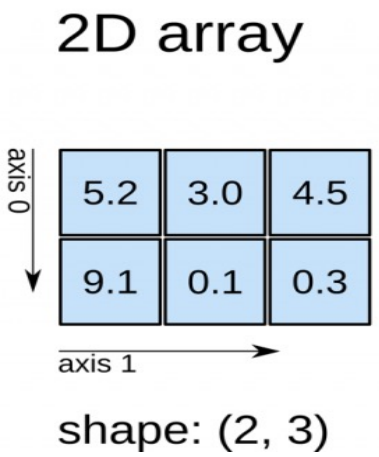
```
L11=v1.tolist() #converts a 1D array to a list
[7, 2, 9, 10]

L22=M.tolist() #converts a 2D array to a list of lists
[[5.2, 3.0, 4.0], [9.1, 0.1, 0.3]]
```

Attributes of an array: shape, size and axis, dtype

```
ndarray.ndim   number of axes, or dimensions, of the array
ndarray.shape  tuple of integers that indicate the number of elements
               stored along each dimension of the array.
ndarray.size   total number of elements of the array.
ndarray.dtype  the type of the elements in the array
```




```
print(v1)
[7 2 9 10]        #vector
print(v1.ndim)    # 1 axis
print(v1.shape)   # (4,) 4 elements in one dimention
print(v1.size)    # 4 elements
print(v1.dtype)   # int64
```




```
print(M)
[[5.2 3.  4.5] #matrix
[9.1 0.1 0.3]]
print(M.ndim)   # 2 axes
print(M.shape)  # (2, 3) 2 rows and 3 columns
print(M.size)   # 6 elements
print(v1.dtype) # float64
```

Create 1D and 2D array of integer random numbers

No need to import the random module

```
np.random.randint(low, high=None, size=None, dtype=int)
```

It generates random integers in range [*low*, *high*).
If *high* is None (the default), then results are from [0, *low*). If size is None returns one value.

```
v=np.random.randint(1,10, size=10)  #1D array, 1 axis
[7 6 3 2 3 9 9 7 4 7]
np.ndim(v)
1

vr=np.random.randint(1,10, size=(1,3)) #2D array, a row vector
[[1 7 4]]

vc=np.random.randint(1,10, size=(3,1)) #2D array, a column vector
[[3]
 [1]
 [1]]

M=np.random.randint(5, size=(2,4)) #2D array, a matrix
[[3 1 4 4]
[4 2 0 4]]
```

look at the doc page https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html

Create 1D and 2D array of real random numbers

To generate random floats in the half-open interval [0.0, 1.0)

```
v1=np.random.random(3)
[0.5769529  0.42219241 0.89814836]  #1D array
```

```
M1=np.random.random((2,3))  #2D array – a matrix
[[0.97055086 0.31126001 0.55647421]
 [0.16726144 0.99078792 0.75163153]]
```

Create 1D array with arange()

np.arange() function is used to generate an array with evenly spaced values within a specified interval. You can define the size step between elements. It returns a 1D array

```
arange(stop)                values generated within [0, stop)
arange(start, stop)         values generated within  [start, stop)
arange(start, stop, step)   values generated within  [start, stop)
with spacing between values given by step.
```

```
a=np.arange(3) #for arrays of integers works as the range function
[0 1 2]

b=np.arange(3,7)
[3 4 5 6]

c=np.arange(1,10,2)
[1 3 5 7 9]

#create 1D array of float
d=np.arange(1,3,0.3)
[1.  1.3 1.6 1.9 2.2 2.5 2.8]

e=np.arange(1,3,0.5)
[1.  1.5 2.  2.5]
```

Create 1D array with linspace()

np.linspace() function is used to create an array with a defined number of elements evenly spaced within a specified range. You can specify the number of elements.
It returns a 1D array

```
linspace(start, stop) creates arrays of 50 (default) evenly spaced
                      numbers over the interval [start, stop]

linspace(start, stop, num=N)   creates N evenly spaced numbers over
                               the interval [start, stop]
```

```
a=np.linspace(0, 1) #create an array of 50 elements in range
[0.1]
```

```
b=np.linspace(2.0, 3.0, num=5) #create an array of 5 elements
in range [2.0, 3.0]
[2.   2.25 2.5  2.75 3.  ]
```

Create 1D and 2D arrays with built-in functions: zeros, eye, ones

```
v = np.zeros((3,)) #tuple (3,) defines the shape, 1D array
print(vr)
[0. 0. 0.] # vector


vc = np.zeros((3,1)) #tuple (3,1) defines the shape 3 rows, 1 column
print(vc)
[[0.]  #column vector
[0.]
[0.]]


M=np.zeros((2, 3)) #tuple (2,3) defines the shape 2 rows, 3 columns
print(M1)
[[0. 0. 0.] #matrix
[0. 0. 0.]]
```

Explore functions ones and eye
https://numpy.org/doc/stable/reference/routines.array-creation.html

Create an array from existing arrays: concatenation

```
np.vstack(tup) Stack arrays in sequence vertically (row wise).
np.hstack(tup) Stack arrays in sequence horizontally (column wise).
tup is sequence type of ndarrays, like a tuple or a list of ndarrays
```

```
A=np.ones((2,3))
[[1. 1. 1.]
[1. 1. 1.]]

B=np.random.randint(10,size=(2,3))
[[5 7 5]
[0 1 5]]

H = np.hstack( (A, B) ) # horizontal concatenation
[[1. 1. 1. 9. 1. 6.]
[1. 1. 1. 0. 1. 4.]]

V = np.vstack( [A, B] ) # vertical concatenation
[[1. 1. 1.]
[1. 1. 1.]
[9. 1. 6.]
[0. 1. 4.]]
```

Shape manipulation: reshape, transpose, flatten

```
v = np.arange(1,11)
[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
```

```
M = np.reshape(v, (2,5))   #change shape of an array
[[ 1  2  3  4  5]
[ 6  7  8  9 10]]
```

```
Mt=np.transpose(M)          #transpose a matrix
[[ 1  6]
[ 2  7]
[ 3  8]
[ 4  9]
[ 5 10]]
```

```
v1=M.flatten()   #method applied to a ndarray returns a 1D array
[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
```

Indexing on ndarrays

ndarrays can be indexed using the standard Python syntax
https://numpy.org/doc/stable/user/basics.indexing.html

`arr[obj]`

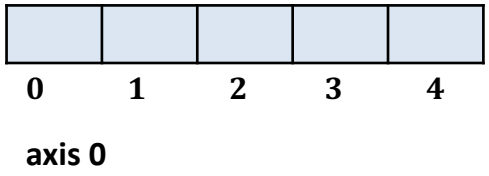
where *arr* is the array and *obj is* the selection.

There are different kinds of indexing available depending on *obj*:

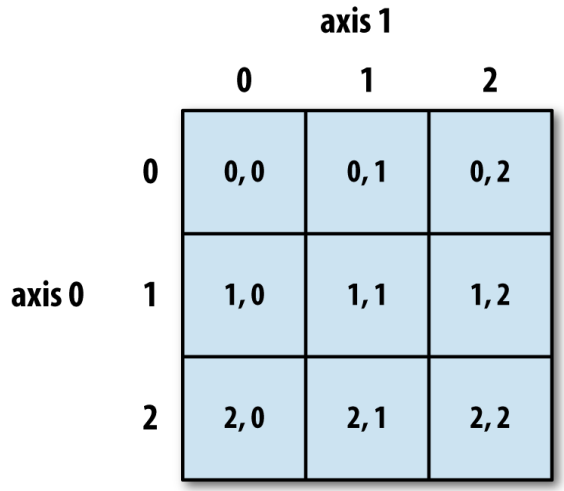
Basic indexing: Single element indexing
Slicing

Advanced indexing: Integer array indexing (Fancy Indexing)
Boolean array indexing (Masking)

1D array



2D array



Basic indexing - 1D array

Single element indexing and slicing work exactly like for other standard Python sequences.

```
arr[index]            # select one element at index
arr[start:end]        # slice from index start through index end-1
arr[start:]           # slice from index start through last index
arr[:end]             # slice from index 0 through index end-1
arr[start:end:step] #slice from index start through not past end by step
```

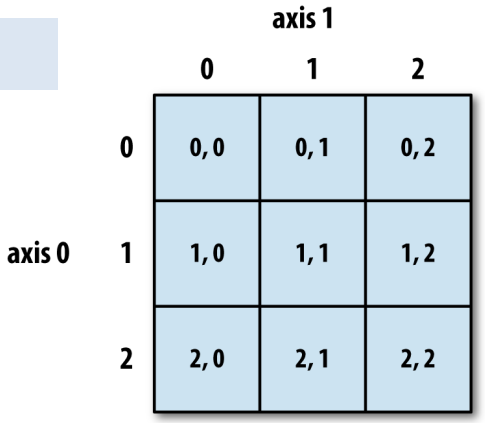
```
arr1 = np.arange(1,11)
[ 1  2  3  4  5  6  7  8  9 10]

arr1[0]   # 1

arr1[-1]  # 10

arr1[3:8] # [4 5 6 7 8]

arr1[5:]  # [ 6  7  8  9 10]

arr1[:5]  # [1 2 3 4 5]

arr1[1:8:2] # [2 4 6 8]
```

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Single element Indexing - 2D array

In a 2D array, to access one element, you need two indices, one for selecting the row and another for selecting the column

```
arr[index_row, index_col]   # select one element
```




```
arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
[[1 2 3]
[4 5 6]
[7 8 9]]
```

```
arr2[0,0] #same as arr2[0][0] as in nested lists
1
```

You can select a specific row

```
arr2[0] #first row
[1 2 3]
```

```
arr2[-1] #last row
[7 8 9]
```

Slicing - 2D array

The standard rules of list slicing apply to basic slicing on a per-dimension basis.

```
arr2d = np.array([[1, 2, 3,4], [5, 6, 7,8], [9, 10, 11,12]])
[[ 1  2  3  4]
[ 5  6  7  8]
[ 9 10 11 12]]
```

```
arr2d[:,1]  #second column
[ 2  6 10]
```

1	2	3	4
5	6	7	8
9	10	11	12

```
arr2d[0,:]  #first row, same as arr2d[0]
[1 2 3 4]
```

1	2	3	4
5	6	7	8
9	10	11	12

```
arr2d[0,::2]
[1 3]
```

1	2	3	4
5	6	7	8
9	10	11	12

```
arr2d[1:,1:]
[[ 6  7  8]
[10 11 12]]
```

1	2	3	4
5	6	7	8
9	10	11	12

Integer array indexing - 1D

Indexing with an integer array or list of integers allows selection of arbitrary items in the array. This method is also called **fancy indexing**. It is like the simple **indexing** we've already seen, but we pass arrays of **indices** in place of single scalars

```
arr[arr_indices]   # selection of multiple arbitrary elements
```

1	2	3	4	**5**	6	**7**	8	9	10
0	1	2	3	**4**	5	**6**	7	8	10

arr1
indices

```
arr1[[0,4,6]] # [0,4,6] is a list of indices
[1 5 7]
```

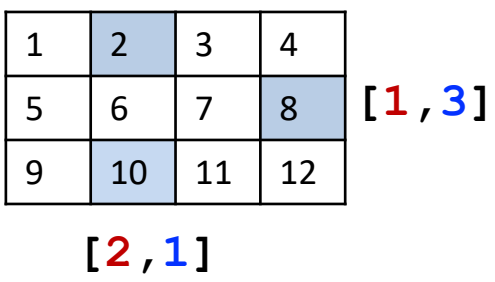
```
indarr=np.array([0,4,6]) #1D array of indices
arr1[indarr]
[1 5 7]
```

Integer array Indexing - 2D array

Integer array Indexing allows selection of arbitrary items in the array.
For 2D array, two integer 1D arrays (or two lists) are needed, one for each dimension.

```
arr[arr_ind_row, arr_ind_col]   # selection of multiple arbitrary elements
```

```
arr2d
[[ 1  2  3  4]
[ 5  6  7  8]
[ 9 10 11 12]]

arr2d[[0,1,2],[1,3,1]]   #provide two lists, one for the rows and another for the columns
[2 8 10]

arr2d[np.array([0,1,2]),np.array([1,3,1])
[2 8 10]
```



Replacing values in 1D array

by using an indexing method on the left side of the equal sign, you can *replace* selected elements of an array.

```
arr[obj]=value
```

The value being assigned to the indexed array must be shape consistent (the same shape or broadcastable to the shape the index produces).

```
arr1=np.arange(1,11)
 [ 1  2  3  4  5  6  7  8  9 10]

arr1[0]=100
[ 100  2  3  4  5  6  7  8  9 10]


arr1[3:7]=12 #a scaler is broadcasted to the entire selection
[ 1  2  3 12 12 12 12  8  9 10]


arr1[5:]=arr1[5:]**2
[  1   2   3   4   5  36  49  64  81 100]
```

Replacing values in 2D array


```
arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
[[1 2 3]
[4 5 6]
[7 8 9]]


arr2[[0,1],[0,1]]=[100,200] #shape consistent
[[100   2   3]
 [  4 200   6]
 [  7   8   9]]


arr2[[0,1],[0,1]]=0 #a scaler is broadcasted to the entire
selection
[[0 2 3]
 [4 0 6]
 [7 8 9]]
```

Summary

Attributes of a NumPy Array

ndarray.ndim	number of axes, or dimensions, of the array
ndarray.shape	tuple of integers that indicate the number of elements stored along each dimension of the array.
ndarray.size	total number of elements of the array.
ndarray.dtype	the type of the elements in the array

type conversion

ndarray.tolist()	method to convert ndarrays to lists
np.array()	function to convert lists to ndarrays

Summary- Create ndarrays using built-in functions

https://numpy.org/doc/stable/reference/routines.array-creation.html

Only 1D array

`np.arange()` function is used to create a 1D array with evenly spaced values within a specified interval.

```
arange(stop)                values generated within [0, stop)
arange(start, stop)         values generated within  [start, stop)
arange(start, stop, step)   values generated within  [start, stop)
with spacing between values given by step.
```

`np.linspace()` function is used to create an array with a defined number of elements evenly spaced within a specified range. You can specify the number of elements.
It returns a 1D array

```
linspace(start, stop)      creates arrays of 50 (default) evenly spaced
                           numbers over the interval [start, stop]

linspace(start, stop, num=N)    creates N evenly spaced numbers over
                                the interval [start, stop]
```

Summary- Create ndarrays using built-in functions

`np.random.randint(low, high=None, size=None, dtype=int)`
generates random integers in range [low, high).
If high is None (the default), then results are from [0, low).

`np.random.random(size=None)` generates random floats in the half-open interval [0.0, 1.0) see also random_samples()

If **size** is None returns one value. To generate a 1D array size=number of elements, to generate a 2D array size=(num rows, num columns)

`np.zeros(shape, dtype=float)` returns a new array of given shape and type, filled with zeros.

`np.ones(shape, dtype=None)` returns a new array of given shape and type, filled with ones.

`np.eye(N, M=None)` returns a 2-D array with ones on the diagonal and zeros elsewhere.

Summary - built-in functions for array manipulation

https://numpy.org/doc/stable/reference/routines.array-manipulation.html

`np.vstack(tup)` Stack arrays in sequence vertically (row wise).
`np.hstack(tup)` Stack arrays in sequence horizontally (column wise).
tup is sequence type of ndarrays, like a tuple or a list of ndarrays

`np.reshape(a, newshape)` returns an array with a new shape without changing its data.

`np.transpose(a)` returns an array with axes transposed.

A method is applied to a ndarray object

`ndarray.flatten()` returns a copy of the array collapsed into one dimension.

Vectorization

Vectorization is one of the core feature of NumPy
Vectorization allows to apply an action to every element of an array with a single line of code, avoiding loops and making your code more readable and efficient.

- Array Operations
- Vectorized functions or ufunc
- Boolean array and indexing

Array Operations

Any arithmetic operations between equal-size arrays applies the operation elementwise

```
A+B      +    element by element addition
A-B      -    element by element subtraction
A*B      *    element by element multiplication
A/B      /    element by element division
A**B     **   element by element power

If is B a scalar it will be broadcasted to all elements of A
```

```
A=np.ones((2,3))
[[1. 1. 1.]
[1. 1. 1.]]

B=np.random.randint(10,size=(2,3))
[[5 4 1]
[4 1 6]]

B*2
[[10 8 2]
[ 8 2 12]]

A+B
[[6. 5. 2.]
[5. 2. 7.]]
```

Vectorized built-in functions

Universal Functions: Fast Element-wise Array Functions
A universal function, or *ufunc*, is a function that performs elementwise operations on data in ndarrays, thus removing the need for flow control loops.

Example of ufuncs are **sqrt, exp, sin, cos, log, log10**.

Available ufunc for math operations can be found here
https://numpy.org/doc/stable/reference/ufuncs.html#math-operations

```
a = np.arange(1,4)
[1 2 3]
```

Calculate the square root of each element in the array a

```
np.sqrt(a)
[1.          1.41421356  1.73205081]
```

```
np.power(a,2)
[1 4 9])
```

Boolean array and indexing

In NumPy you can perform element-wise comparison

```
>
>=
<
<=
==
!=
```

```
logical_and, logical_or
equivalent to operators & |
```

Comparison operators applied to an array, return a Boolean array of the same size.

```
arr1=np.arange(1,11)
 [ 1  2  3  4  5  6  7  8  9 10]

bool1=arr1 >= 6
[False False False False False  True  True  True  True  True]
```

1	2	3	4	5	6	7	8	9	10
False	False	False	False	False	True	True	True	True	True

```
arr1[bool1]  #extract True elements
[ 6 7  8 9 10 ]

arr1[bool1]=arr1[bool1]*10 #replace True elements
[ 1 2 3 4 5 60 70 80 90 100 ]
```

Boolean array and indexing


```
bool2 = np.logical_and( arr1 < 8 , arr1 >= 5 )
[False False False False  True  True  True False False]

arr1[bool2]
[5 6 7]
```

```
bool3 = np.logical_or( arr1 <=3 , arr1 > 9 )

arr1[bool3]
[ 1  2  3 10]
```

How would you count the elements satisfying condition?

Also explore the function np.count_nonzero()
https://numpy.org/doc/stable/reference/generated/numpy.count_nonzero.html

Boolean array and indexing


```
arr2=np.array([[1,2,3,4],[5,6,7,8]])
[[1 2 3 4]
[5 6 7 8]]

bool1=np.logical_or(arr2 <=2, arr2 > 6)
[[ True  True False False]
[False False  True  True]]


arr2[bool1]  #extract True  elements
[1 2 7 8]



arr2[bool1]=arr2[bool1]**2 #replace True elements
[[ 1  4  3  4]
 [ 5  6 49 64]]

arr2[bool1]=0
[[0 0 3 4]
 [5 6 0 0]]
```

Finding maximum and minimum - max, min, argmax, argmin

```
M=np.max(A, axis=n)       return the maximum along a given axis.
m=np.min(A, axis=n)       return the minimum along a given axis.
ind=np.argmax(A, axis=n) return the indices of the maximum values along
an axis.
ind=np.argmix(A, axis=n) return the indices of the minimum values along
an axis.
```

```
M = np.array([[1, 2, 3,4], [8, 6, 0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
[ 8  6  0  9]
[ 9  2  3 12]]
```

```
axis=1   row
axis=0   column
```

```
vm=np.min(M, axis=1)        #returns minimum of each row
[1 0 2]

indm=np.argmin(M, axis=1) #returns the indices of the minimum
[0, 2, 1]                  values of each row
```

In the case of a 1D array, no need to specify the axis

```
v1=np.random.randint(1,50, size=6)
[22, 38, 30, 40, 18,  1]

m=np.min(v1) #1  returns the minimum value
i=np.argvmin(v1) #5  returns the index of the min value
```

Sorting - sort, argsort

S=np.sort(A, axis=n) return a sorted copy of an array. Sort along an axis.
ind=np.argsort(A, axis=n) returns an array of indices of the same shape as *A* that index elements along the given axis in sorted order

```
Ms=np.sort(M, axis=0)  #sort each column
[[ 1  1  0  4]
[ 8  2  3  9]
[ 9  6  3 12]]

Mind=np.argsort(M, axis=0) #Returns indices that would sort M
[[0 2 1 0]
[1 0 0 1]
[2 1 2 2]]


vs=np.sort(v1)  #returns the sorted array
[ 1, 18, 22, 30, 38, 40]

ind=np.argsort(v1) #returns the indices that would sort an
array
[5, 4, 0, 2, 1, 3]
```

some statistical functions – mean, std

```
m=np.mean(A, axis=n)  compute and return the arithmetic mean along
the specified axis.

s=np.std(A, axis=n)   compute and return the standard deviation
along the specified axis.
```

```
M = np.array([[1, 2, 3,4], [8, 6, 0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
[ 8  6  0  9]
[ 9  2  3 12]]

m=np.mean(M, axis=0) #mean of each column
[6.         3.33333333 2.         8.33333333]

s=np.std(M, axis=1) #std of each row
[1.11803399 3.49106001 4.15331193]
```

More statistical functions
https://numpy.org/doc/stable/reference/routines.statistics.html

Some math functions – sum, cumsum

```
s=sum(A, axis=n)    calculate and return the sum of array elements
over a given axis.

c=cumsum(A, axis=n) return the cumulative sum of the elements along a
given axis.
```

```
M = np.array([[1, 2, 3,4], [8, 6,
0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
[ 8  6  0  9]
[ 9  2  3 12]]

s=np.sum(M, axis=0) #sum of each column
[18 10  6 25]

c=np.cumsum(M, axis=1) #cumulative sum of each row
[[ 1  3  6 10]
 [ 8 14 14 23]
 [ 9 11 14 26]]
```

More math functions
https://numpy.org/doc/stable/reference/routines.math.html

Importing data with loadtxt:

```
A=np.loadtxt(filename, delimiter='sep')  store data in a 2D array
```

- loadtxt will work on ASCII files that contain **numbers**
- If the field separator is not a space, you should specify a delimiter character

Example: file is on Canvas DATA

```
dat = np.loadtxt('ph.dat')
```

You can use slicing to store the first column in variable ph, and second column in variable p:

```
ph = dat[:,0]
p = dat[:,1]
```

You can also define the dtype of the resulting array, and skip comment lines
https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html

Exporting array with savetxt

`np.savetxt(filename, X, fmt='format', delimiter='sep')` save an array to a text file.

If you do not specify the fmt and delimiter, it will format as %.18e with a space as delimiter

```
M = np.array([[1, 2, 3,4], [8,6, 0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
[ 8  6  0  9]
[ 9  2  3 12]]
```

```
np.savetxt('file1.txt', M, fmt='%d', delimiter=':') #save in file1.txt, format each number as integer, and set delimiter to colon.
```

```
np.savetxt('file1.csv', M, fmt='%.1f', delimiter=',') #save in file1.csv, format each number as float with 1 decimal precision, and set delimiter to comma.
```

You can also write the header, and comment lines
https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html

Looping over ndarrays using the usual python syntax

Numpy arrays are iterables and so you can loop over its elements with the usual python syntax.

Iterate on the elements of a 1-D array:

```
arr = np.array([1, 2, 3])
for x in arr:
    print(x)
```

```
1
2
3
```

In a 2-D array the standard for loop will go through all the rows.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
  print(x)
```

```
[1 2 3]
[4 5 6]
```

To iterate on each scalar element of the 2-D array, see next slide.

Looping over elements of ndarrays – np.nditer()

To iterate through each scalar (element) of a ndarray we need to use n for loops, where n is the dimension of the array.

To avoid nested loops, you can use the **nditer()**
https://numpy.org/doc/stable/reference/arrays.nditer.html

nditer() iterates through each scalar element of a ndarray (one by-one).

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in np.nditer(arr):
  print(x)
```

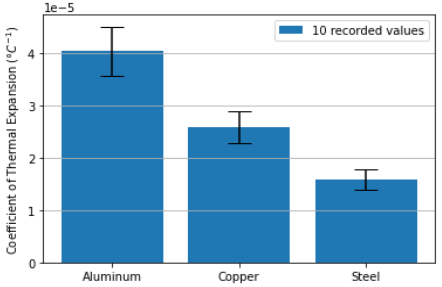
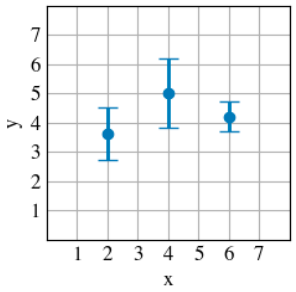
```
1
2
3
4
5
6
```

Summary of built-in functions useful for data analysis

```
B=np.function(A, axis=n)

axis=0: apply operation column-wise, across all rows for each column.
axis=1: apply operation row-wise, across all columns for each row
```

`M=np.max(A, axis=n)`	maximum along a given axis.
`ind=np.argmax(A, axis=n)`	indices of the maximum values along an axis.
`m=np.min(A, axis=n)`	minimum along a given axis.
`ind=np.argmix(A, axis=n)`	indices of the minimum values along an axis
`S=np.sort(A, axis=n)`	sorted copy of an array. Sort along an axis.
`ind=np.argsort(A, axis=n)`	indices in sorted order
`m=np.mean(A, axis=n)`	mean along the specified axis.
`s=np.std(A, axis=n)`	standard deviation along the specified axis.
`s=sum(A, axis=n)`	sum of elements over a given axis.
`c=cumsum(A, axis=n)`	cumulative sum of elements along a given axis.

Visualizing errors

`ax.bar(x,mean_values, yerr=stderr)`	bar graph with error bar	compare data and errors between different groups	
`ax.`**`errorbar`**`(x,y,yerr)`	error bar	Visualize data and errors	

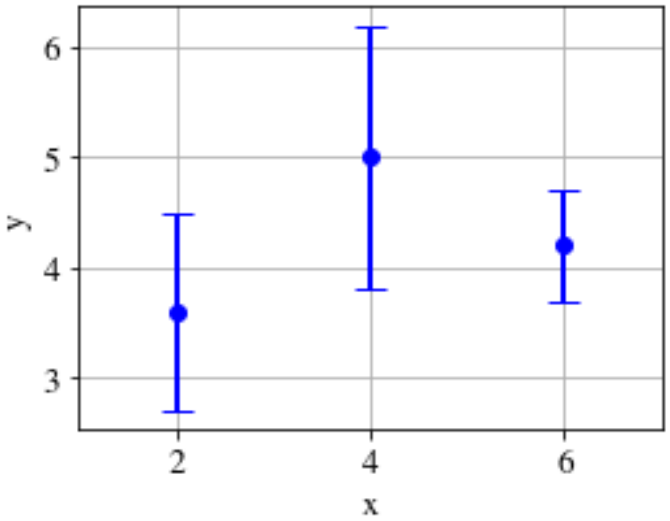
The error bar usually reports the standard error, where σ is the standard deviation, and N the number of samples

$$SE = \frac{\sigma}{\sqrt{N}}$$

Look at the documentation page of **errorbar()**
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.errorbar.html
and **bar()** to make a bar graph with error bars
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.bar.html

Example – errorbar()

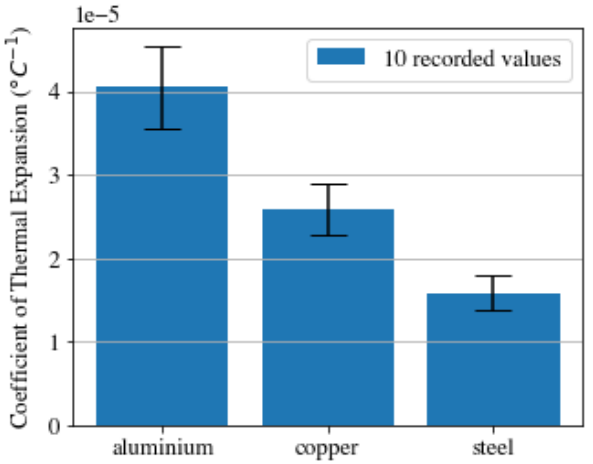
```
import matplotlib.pyplot as plt
import numpy as np
plt.rcParams['font.size']=13

#some fictitious data from a fictitious experiment.
x = [2, 4, 6]
y = [3.6, 5, 4.2]
yerr = [0.9, 1.2, 0.5]

# plot
fig, ax = plt.subplots(figsize=(4,3))
ax.errorbar(x, y, yerr, fmt='o', linewidth=2, capsize=6, color='b')
ax.set_xlim(1,7)
ax.ylim=(0, 8)
ax.xticks=np.arange(1, 8)
ax.yticks=np.arange(1, 8)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid()
plt.show()
```



Example – bar graph with error bars

We use the measured coefficient of thermal expansion (CTE) of three metals: Aluminum, Copper, and Steel. The unit for coefficient of thermal expansion is per degrees Celsius (/ °C). We calculate the mean and standard error for each metal and use a bar graph where we also visualize the standard error.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

df=pd.read_csv('data-metals.csv') # find the data set on Canvas/DATA
print(df)

#calculate the average for each metal/column
mean_metal=df.mean(axis=0)

#calculate the standar error for each metal/column
std=df.std(axis=0)
std_error=std/np.sqrt(len(df))

#make the bar graph with error bars
fig, ax = plt.subplots(figsize=(5,4))
plt.rcParams['font.size']=13
names=df.columns.values
ax.bar(names,mean_metal, yerr=std_error, capsize=10, label='10 recorded
values')
ax.set_ylabel('Coefficient of Thermal Expansion ($\degree C^{-1}$)')
ax.grid(axis='y')
ax.legend()
```



Matplotlib

Matplotlib is a commonly used libraries, that provide various tools for data visualization in Python. Matplotlib is based on NumPy.

Matplotlib works on ndarrays, but also on lists and tuples of numbers, and pandas objects.

Import Matplotlib, its submodule Pyplot

```
import matplotlib.pyplot as plt
```

There are two Matplotlib Interfaces: the Object-Oriented Interface and the Functional Interface.
In this course we will cover the Object-Oriented Interface

Object Oriented Interface

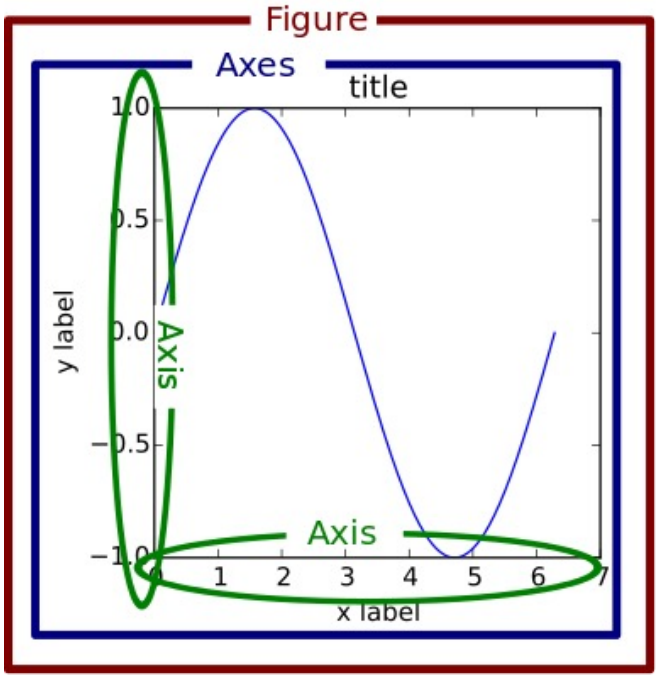
```
fig, ax = plt.subplots() #creates one Figure Object and one Axes Object

Apply methods to the Axes Object
#some plotting methods
    ax.plot()
    ax.bar()

#Label the axis:
    ax.set_xlabel()
    ax.set_ylabel()

# Show the grid:
     ax.grid()

#Setting x and y limits
    ax.set_xlim()
    ax.set_ylim()

#Setting x and y ticks
    ax.set_yticks()
    ax.set_xticks()

#Show the legend
   ax.legend()
```

Show the graph:

```
plt.show()
```

Apply methods to the fig object.

```
fig.savefig('figurename.png') #save figure in png
```



Do not confuse the x axis, and y axis with the Axes Object. The x axis and y axis are elements of the Axes Object.

Object Oriented Interface

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 10, 0.1)
y = np.sin(np.pi * x) + x


fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlabel("x")
ax.set_ylabel("y")
```

Functional Interface

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0,10,0.1)
y = np.sin(np.pi*x) + x


plt.figure()
plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("y")
```

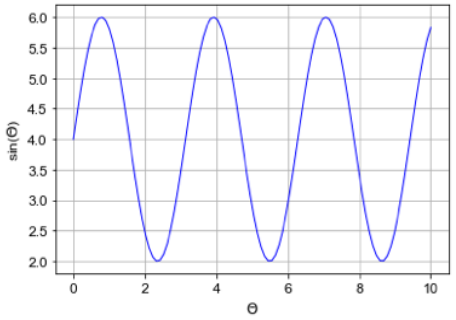
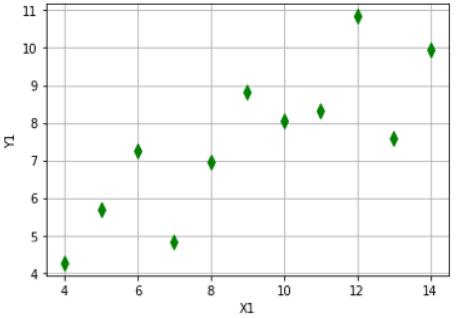
OO Interface useful links:
https://matplotlib.org/stable/gallery/showcase/anatomy.html

There are very helpful Matplotlib Cheatsheets and Handouts you can use:
https://matplotlib.org/cheatsheets/

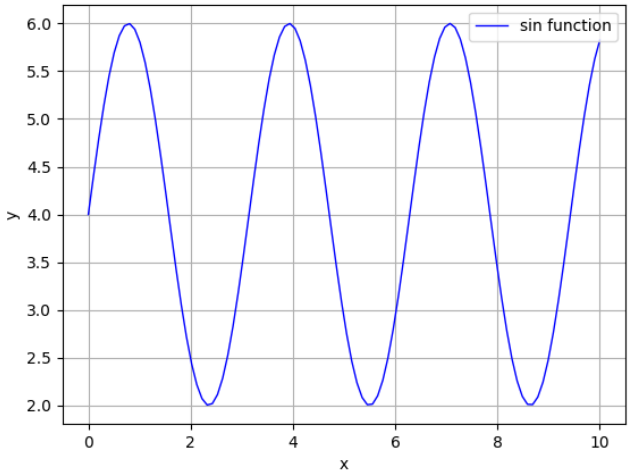
One useful handout for beginners can be found here:
https://matplotlib.org/cheatsheets/_images/handout-beginner.png

The plot() method

https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.plot.html

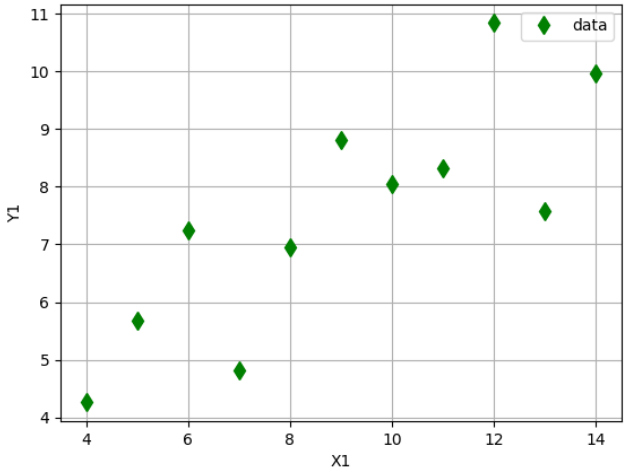
Axes method	plot type	use	figure plot
`ax.plot(x, y, '-b')`	Line plot	track changes over time, visualize mathematical functions	
`ax.plot(x, y, 'dg')`	plot with markers – scatterplot	Visualize possible relationships between two parameters, Visualize experimental data	

line plot




```
# make data
x = np.linspace(0, 10, 100)
y = 4 + 2 * np.sin(2 * x)

fig, ax = plt.subplots() #Create one Figure object fig and one Axes
object ax

#apply methods to the Axes object ax
ax.plot(x, y, '-b', linewidth=1, label='sin function') #blue solid
line is defined by the string '-b'

ax.set_xlabel('x')  #label x axis
ax.set_ylabel('y')  #label y axis

ax.grid()   #show the grid
ax.legend() #show the legend. Text is reported in the label of plot
plt.show()  #show the plot. Notice show is applied to plt.
```

scatter plot




```
import matplotlib.pyplot as plt
import numpy as np


x=np.array([10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0, 5.0])
y=np.array([8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82,
5.68])


fig, ax = plt.subplots()
ax.plot(x, y, 'dg', markersize=8, label='data') #string 'dg' sets a green
diamond marker
ax.set_xlabel("X1")
ax.set_ylabel("Y1")

ax.grid()
ax.legend()
plt.show()
```

Specify line, marker, color and type in plot()

```
ax.plot(x, y, 'ob-', linewidth=1, markersize=8) #example
```

A **format string** is a string type containing characters that specify markers, lines and colors.
A **format string** is given by **'marker line color'** #each of them is optional

Example format strings:

```
'og'    # green circles
'-b'    # blue solid line
'--'    # dashed line with default color
'^k:'   # black triangle up markers connected by a dotted line
```

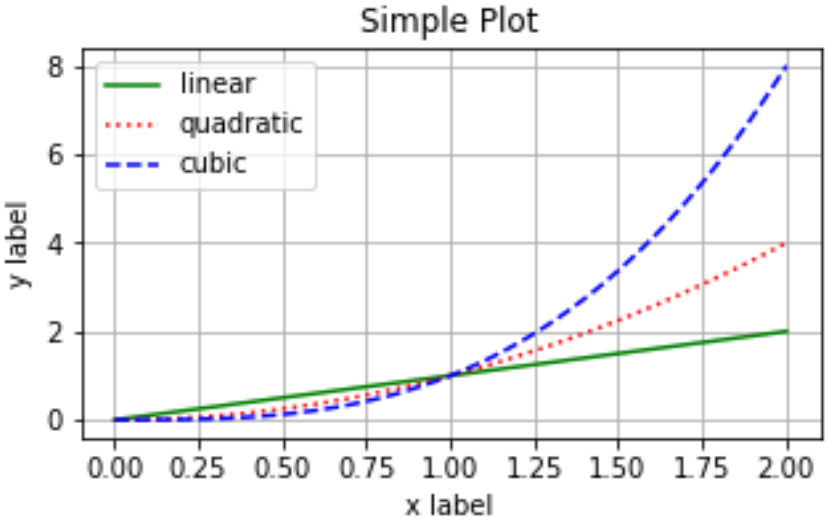
Here we report some characters for the format string

Markers		Line Styles		color	
.	point marker	-	solid line style	b	blue
o	circle marker	--	dashed line style	g	green
v	triangle down marker	-.	dash-dot line style	r	red
s	square marker	:	dotted line style	k	black
*	star marker				
D	diamond marker				

https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.plot.html

Multiple plots on the same Axes

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 100)
```




```
fig, ax = plt.subplots(figsize=(5, 2.7)) # figure size in inches
ax.plot(x, x, '-g', label='linear')      #Plot some data on the Axes.
ax.plot(x, x**2, ':r', label='quadratic') #Plot more data on the same Axes
ax.plot(x, x**3, '--b', label='cubic') #Plot more data on the same Axes.

ax.set_xlabel('x label')
ax.set_ylabel('y label')


ax.set_title("Simple Plot")
ax.legend() #Add a legend
ax.grid()
plt.show()
```

set x and y limits and x and y ticks

```
ax.set_xlim(0, 12)   #sets x axis limit in the data coordinate
ax.set_ylim(0, 12)

ax.set_yticks(np.arange(0,13)) #setting x ticks, array of
tick's location

ax.set_xticks(np.arange(0,13))
```

set the figure size, and save the figure

```
fig, ax = plt.subplots(figsize=(5, 2.7)) #figure size in inches
```

```
fig.savefig('test', dpi=300) #saves figure with 300 dpi, by
default as .png
```

```
image_format = 'eps' # e.g .png, .svg, etc.
image_name = 'myimage.eps'

fig.savefig(image_name, format=image_format, dpi=1200)
```

Write Mathematical Expressions and Greek Symbols

You can use a subset of TeX markup in any Matplotlib text string by placing it inside a pair of dollar signs ($).

Any text element can use math text. You should use raw strings (precede the quotes with an 'r'), and surround the math text with dollar signs ($), as in TeX

```
ax.set_title(r'$\alpha > \beta$')
```

$\alpha > \beta$

https://matplotlib.org/stable/users/explain/text/mathtext.html

rcParams to change default settings – e.g. change font size, font name

https://matplotlib.org/stable/users/explain/customizing.html

rcParams is a like-dictionary type, storing different settings.

The see the default settings type:

```
import matplotlib.pyplot as plt
print(plt.rcParams)
```

```
#the rcParams must be placed before the plt.subplots() to work
```

To change setting, overwrite a value, like in a dictionary

```
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.linestyle'] = '--'
plt.rcParams['font.size']=12
plt.rcParams['font.family']='Ariel'
plt.rcParams['figure.subplot.wspace']= 0.4 #set width of the
padding between subplots

plt.rcParams['figure.subplot.hspace']= 0.3 #set the height of
the padding between subplots
```

Set the legend

There are different ways to make a legend, and here we will go over one of them, which is the automatic detection of elements to be shown in the legend taken from the label parameter of the plotting methods.

```
ax.plot(x, x, '-g', label='linear')
ax.legend()    # detect the string of the label parameter of
the plot method and construct the legend.
```

In this example, we have multiple plotting methods on the same Axes.

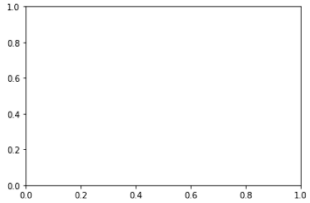
```
ax.plot(x, x, '-g', label='linear')
ax.plot(x, x**2, ':r' label='quadratic')
ax.plot(x, x**3, '--b', label='cubic')
ax.legend()    # detect each string of each label in the plot
methods and construct the legend.
```

subplots

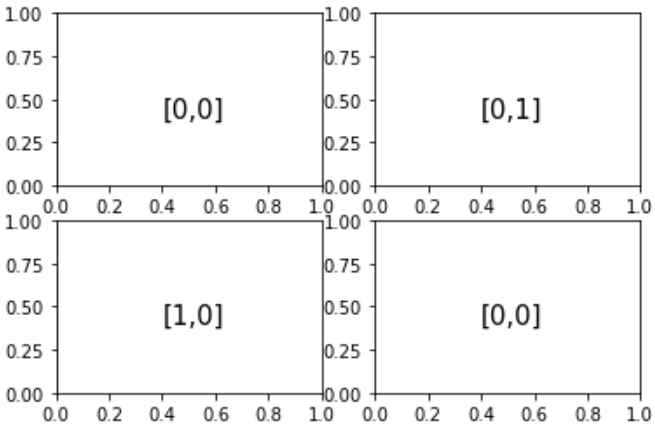
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html

```
plt.subplots(nrows, ncols) # default nrows=1 and ncols=1
```

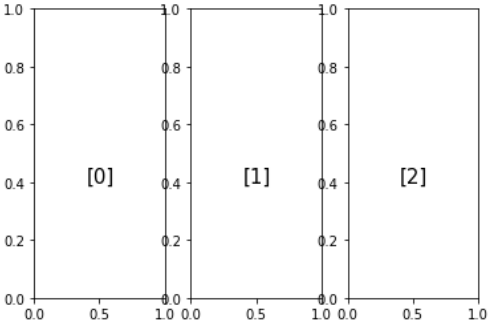
```
fig, ax = plt.subplots() # make a single Axes object
```



```
#make a 2x2 matrix of Axes Objects,
referenced by axs
fig, axs = plt.subplots(2, 2)
axs[0,0].plot()  #apply to Axes [0,0]
axs[0,0].set_xlabel()

axs[0,1].plot() #apply to Axes [0,1]
```

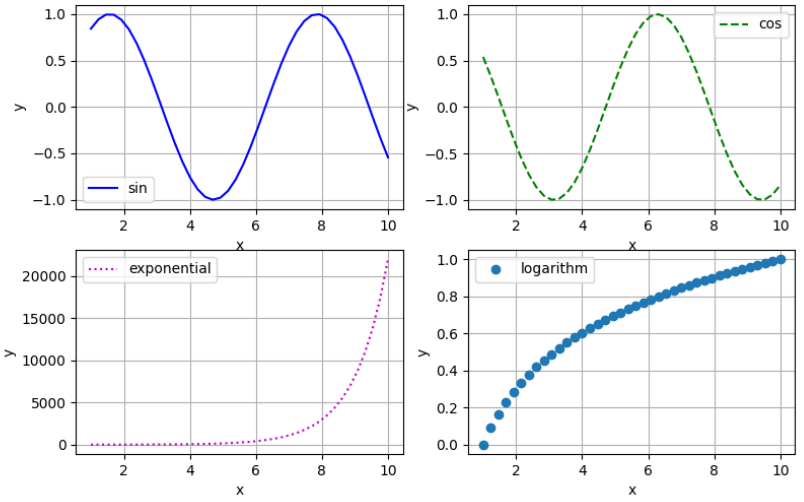



```
#make a row vector of 3 Axes Objects,
referenced by axs
fig, axs= plt.subplots(1, 3)
axs[0].bar() # apply to Axes [0]
axs[1].bar() # apply to Axes [1]
```



Subplot example

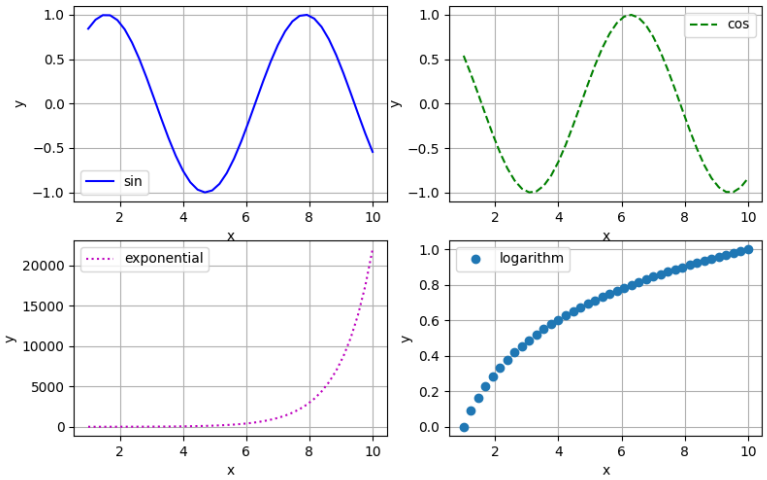

```
import matplotlib.pyplot as plt
import numpy as np
fig,axs =  plt.subplots(2,2, figsize=(12, 12))
x = np.linspace(1,10,num=40)

#Apply method to Axes element [0,0]
axs[0,0].plot(x,np.sin(x),'-b',label='sin')
axs[0,0].set_xlabel('x')
axs[0,0].set_ylabel('y')
axs[0,0].grid()
axs[0,0].legend()

#Apply method to Axes element [0.1]
axs[0,1].plot(x,np.cos(x),'--g', label='cos')
axs[0,1].set_xlabel('x')
axs[0,1].set_ylabel('y')
axs[0,1].grid()
axs[0,1].legend()

#Apply method to Axes element [1,0]
axs[1,0].plot(x,np.exp(x),':m',label='exponential')
axs[1,0].set_xlabel('x')
axs[1,0].set_ylabel('y')
axs[1,0].grid()
axs[1,0].legend()

#Apply method to Axes element [1,1]
axs[1,1].plot(x,np.log10(x),'o', label='logarithm')
axs[1,1].set_xlabel('x')
axs[1,1].set_ylabel('y')
axs[1,1].grid()
axs[1,1].legend()
plt.show()
```



Let's make the same subplots by using a for loop →

Use a loop to make subplots


```
import matplotlib.pyplot as plt
import numpy as np

fig,axs =  plt.subplots(2,2, figsize=(12, 12))
x = np.linspace(1,10,num=40)


#make lists that contain data we use within the loop
formats=['-b','g--', ':m','o'] # format strings for the plot
labels=['sin','cos','exponential','logarithm']
func=[np.sin,np.cos,np.exp,np.log10] # ufunc


for i,j in enumerate(axs.flatten()): # we flat the matrix of Axes, i is the
index and j is the Axes element
      j.plot(x,func[i](x),formats[i],label=labels[i])
      j.set_xlabel('x')
      j.set_ylabel('y')
      #j.set_xlim(1,10)
      #j.set_xticks(np.arange(1,10))
      j.grid()
      j.legend()

plt.show()
```



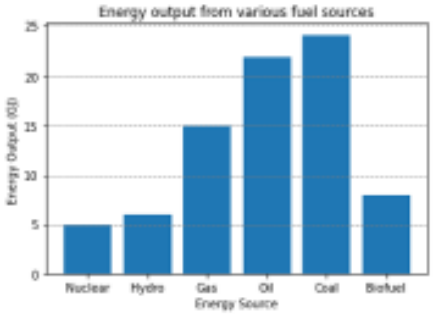
Create a custom plotting function


```
import matplotlib.pyplot as plt
import numpy as np
```

This function takes parameters: the Axes, etc, and returns the customized Axes

```
def myplot(ax0,xvalue,yvalue,formatstr, labelx, labely):
    ax0.plot(xvalue, yvalue,formatstr)
    ax0.set_xlabel(labelx)
    ax0.set_ylabel(labely)
    return ax0


x = np.linspace(1,10,num=40)
y=np.sin(x)


fig, ax = plt.subplots()
ax_out=myplot(ax,x,y,'--b', 'x','y') #call the function
plt.show()

fig, ax1 = plt.subplots()
y1=np.cos(x)
ax_out1=myplot(ax1,x,y1,'--r', 'x','y') # call the function
ax_out1.set_title("popolo") # you can apply methods to the returned
object
plt.show()
```

The bar graph

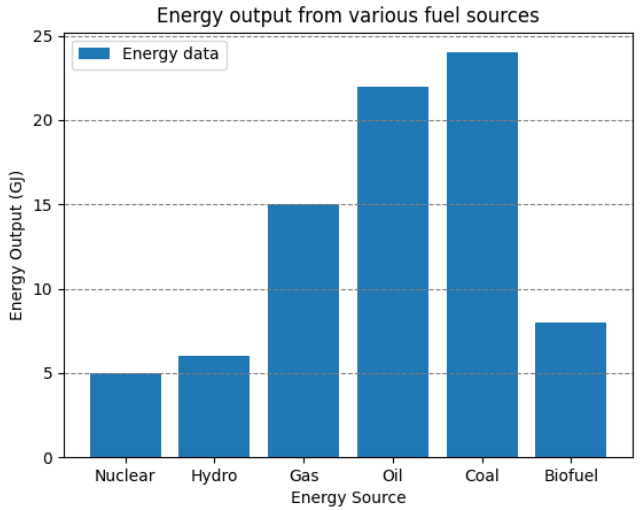
`ax.bar(x,energy)`	bar graph	compare data between different groups	

https://matplotlib.org/stable/gallery/lines_bars_and_markers/index.html

You can make a bar graph by using 1D arrays, lists or Series

Making a bar graph by using Series

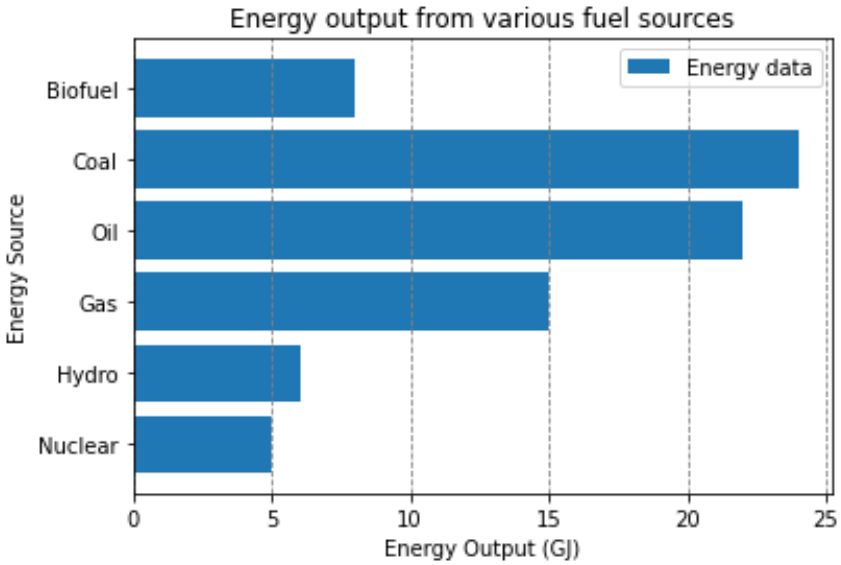

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np


x = ['Nuclear', 'Hydro', 'Gas', 'Oil',
     'Coal', 'Biofuel']
y = [5, 6, 15, 22, 24, 8]
```




```
#We can create a pandas DataFrame
energy_df = pd.DataFrame({'Source': x, 'Energy Production': y})
source = energy_df['Source'] # Series
energy = energy_df.loc[:,'Energy Production'] # Series

fig, ax = plt.subplots()
ax.bar(source, energy, label="Energy data")
ax.set_xlabel("Energy Source")
ax.set_ylabel("Energy Output (GJ)")

ax.set_title("Energy output from various fuel sources") #Add title

# we set the grid to be only horizontal
ax.grid(visible=True, which='major', axis='y', color='grey', linestyle='--')
ax.legend()
plt.show()
```

The horizontal bar graph

To make a horizontal bar graph use barh(y,x)
In barh(y,x) the first argument (y) is plotted on the y axis, and the second (x) on the x axis.



Matplotlib

Matplotlib is a commonly used libraries, that provide various tools for data visualization in Python. Matplotlib is based on NumPy.

Matplotlib works on ndarrays, but also on lists and tuples of numbers, and pandas objects.

Import Matplotlib, its submodule Pyplot

```
import matplotlib.pyplot as plt
```

There are two Matplotlib Interfaces: the Object-Oriented Interface and the Functional Interface.
In this course we will cover the Object-Oriented Interface

Object Oriented Interface

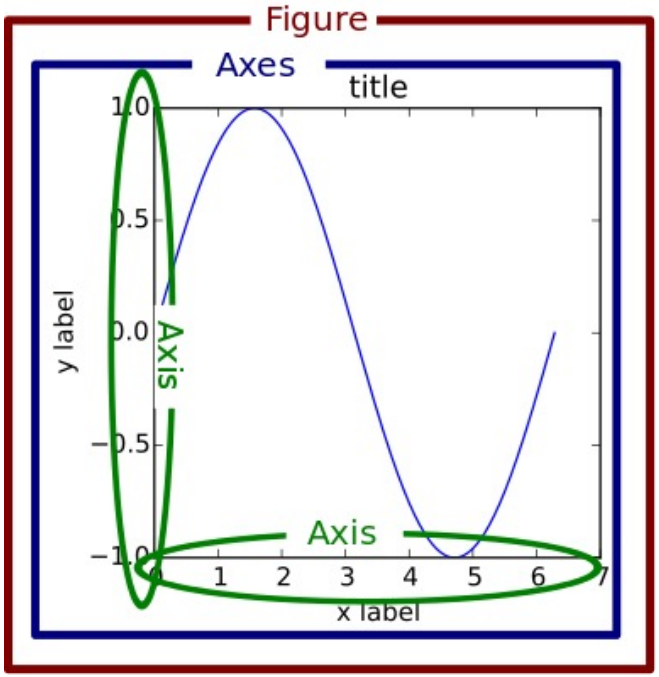
```
fig, ax = plt.subplots() #creates one Figure Object and one Axes Object

Apply methods to the Axes Object
#some plotting methods
    ax.plot()
    ax.bar()

#Label the axis:
    ax.set_xlabel()
    ax.set_ylabel()

# Show the grid:
     ax.grid()

#Setting x and y limits
    ax.set_xlim()
    ax.set_ylim()

#Setting x and y ticks
    ax.set_yticks()
    ax.set_xticks()

#Show the legend
   ax.legend()
```

Show the graph:

```
plt.show()
```

Apply methods to the fig object.

```
fig.savefig('figurename.png') #save figure in png
```



Do not confuse the x axis, and y axis with the Axes Object. The x axis and y axis are elements of the Axes Object.

Object Oriented Interface

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 10, 0.1)
y = np.sin(np.pi * x) + x


fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlabel("x")
ax.set_ylabel("y")
```

Functional Interface

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0,10,0.1)
y = np.sin(np.pi*x) + x


plt.figure()
plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("y")
```

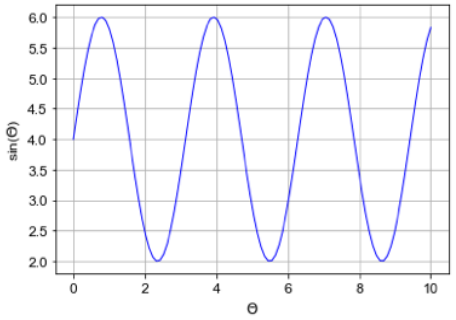
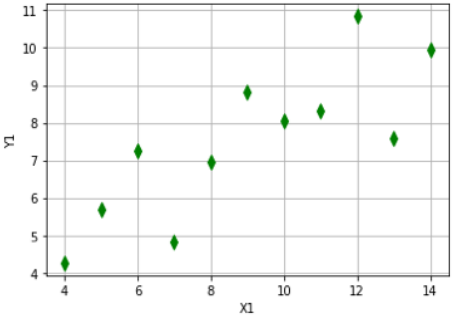
OO Interface useful links:
https://matplotlib.org/stable/gallery/showcase/anatomy.html

There are very helpful Matplotlib Cheatsheets and Handouts you can use:
https://matplotlib.org/cheatsheets/

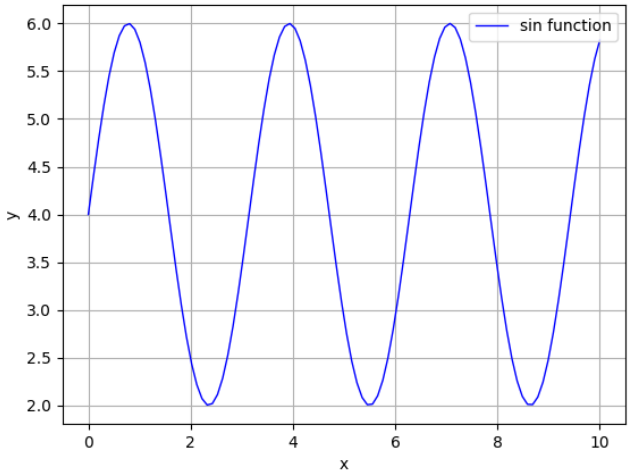
One useful handout for beginners can be found here:
https://matplotlib.org/cheatsheets/_images/handout-beginner.png

The plot() method

https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.plot.html

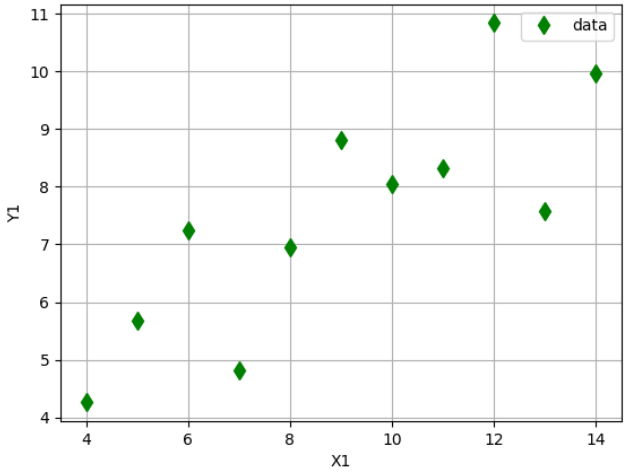
Axes method	plot type	use	figure plot
`ax.plot(x, y, '-b')`	Line plot	track changes over time, visualize mathematical functions	
`ax.plot(x, y, 'dg')`	plot with markers – scatterplot	Visualize possible relationships between two parameters, Visualize experimental data	

line plot




```
# make data
x = np.linspace(0, 10, 100)
y = 4 + 2 * np.sin(2 * x)

fig, ax = plt.subplots() #Create one Figure object fig and one Axes
object ax

#apply methods to the Axes object ax
ax.plot(x, y, '-b', linewidth=1, label='sin function') #blue solid
line is defined by the string '-b'

ax.set_xlabel('x')  #label x axis
ax.set_ylabel('y')  #label y axis

ax.grid()   #show the grid
ax.legend() #show the legend. Text is reported in the label of plot
plt.show()  #show the plot. Notice show is applied to plt.
```

scatter plot




```
import matplotlib.pyplot as plt
import numpy as np


x=np.array([10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0, 5.0])
y=np.array([8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82,
5.68])


fig, ax = plt.subplots()
ax.plot(x, y, 'dg', markersize=8, label='data') #string 'dg' sets a green
diamond marker
ax.set_xlabel("X1")
ax.set_ylabel("Y1")

ax.grid()
ax.legend()
plt.show()
```

Specify line, marker, color and type in plot()

```
ax.plot(x, y, 'ob-', linewidth=1, markersize=8) #example
```

A **format string** is a string type containing characters that specify markers, lines and colors.
A **format string** is given by **'marker line color'** #each of them is optional

Example format strings:

```
'og'    # green circles
'-b'    # blue solid line
'--'    # dashed line with default color
'^k:'   # black triangle up markers connected by a dotted line
```

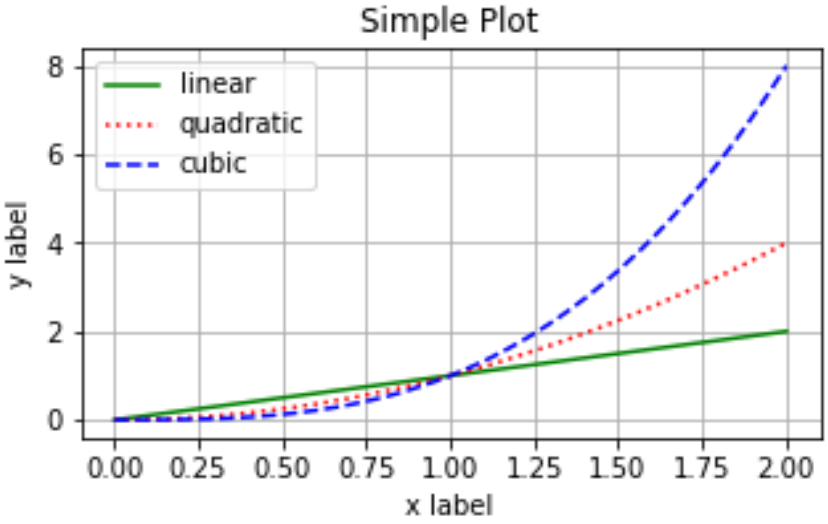
Here we report some characters for the format string

Markers		Line Styles		color	
.	point marker	-	solid line style	b	blue
o	circle marker	--	dashed line style	g	green
v	triangle down marker	-.	dash-dot line style	r	red
s	square marker	:	dotted line style	k	black
*	star marker				
D	diamond marker				

https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.plot.html

Multiple plots on the same Axes

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 100)
```




```
fig, ax = plt.subplots(figsize=(5, 2.7)) # figure size in inches
ax.plot(x, x, '-g', label='linear')      #Plot some data on the Axes.
ax.plot(x, x**2, ':r', label='quadratic') #Plot more data on the same Axes
ax.plot(x, x**3, '--b', label='cubic') #Plot more data on the same Axes.

ax.set_xlabel('x label')
ax.set_ylabel('y label')


ax.set_title("Simple Plot")
ax.legend() #Add a legend
ax.grid()
plt.show()
```

set x and y limits and x and y ticks

```
ax.set_xlim(0, 12)   #sets x axis limit in the data coordinate
ax.set_ylim(0, 12)

ax.set_yticks(np.arange(0,13)) #setting x ticks, array of
tick's location

ax.set_xticks(np.arange(0,13))
```

set the figure size, and save the figure

```
fig, ax = plt.subplots(figsize=(5, 2.7)) #figure size in inches
```

```
fig.savefig('test', dpi=300) #saves figure with 300 dpi, by
default as .png
```

```
image_format = 'eps' # e.g .png, .svg, etc.
image_name = 'myimage.eps'

fig.savefig(image_name, format=image_format, dpi=1200)
```

Write Mathematical Expressions and Greek Symbols

You can use a subset of TeX markup in any Matplotlib text string by placing it inside a pair of dollar signs ($).

Any text element can use math text. You should use raw strings (precede the quotes with an 'r'), and surround the math text with dollar signs ($), as in TeX

```
ax.set_title(r'$\alpha > \beta$')
```

$\alpha > \beta$

https://matplotlib.org/stable/users/explain/text/mathtext.html

rcParams to change default settings – e.g. change font size, font name

https://matplotlib.org/stable/users/explain/customizing.html

rcParams is a like-dictionary type, storing different settings.

The see the default settings type:

```
import matplotlib.pyplot as plt
print(plt.rcParams)
```

```
#the rcParams must be placed before the plt.subplots() to work
```

To change setting, overwrite a value, like in a dictionary

```
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.linestyle'] = '--'
plt.rcParams['font.size']=12
plt.rcParams['font.family']='Ariel'
plt.rcParams['figure.subplot.wspace']= 0.4 #set width of the
padding between subplots

plt.rcParams['figure.subplot.hspace']= 0.3 #set the height of
the padding between subplots
```

Set the legend

There are different ways to make a legend, and here we will go over one of them, which is the automatic detection of elements to be shown in the legend taken from the label parameter of the plotting methods.

```
ax.plot(x, x, '-g', label='linear')
ax.legend()    # detect the string of the label parameter of
the plot method and construct the legend.
```

In this example, we have multiple plotting methods on the same Axes.

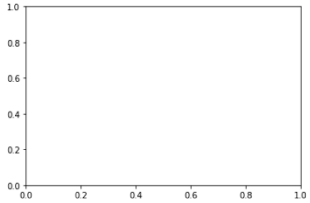
```
ax.plot(x, x, '-g', label='linear')
ax.plot(x, x**2, ':r' label='quadratic')
ax.plot(x, x**3, '--b', label='cubic')
ax.legend()    # detect each string of each label in the plot
methods and construct the legend.
```

subplots

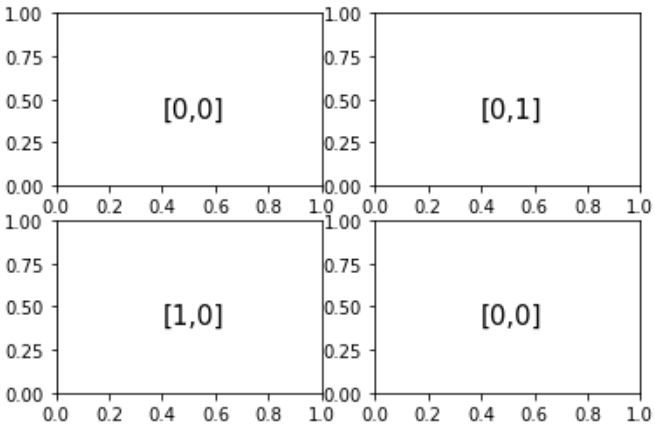
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html

```
plt.subplots(nrows, ncols) # default nrows=1 and ncols=1
```

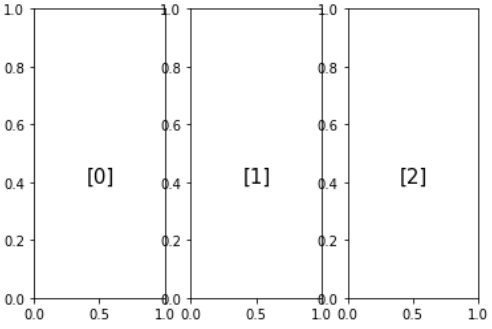
```
fig, ax = plt.subplots() # make a single Axes object
```



```
#make a 2x2 matrix of Axes Objects,
referenced by axs
fig, axs = plt.subplots(2, 2)
axs[0,0].plot()  #apply to Axes [0,0]
axs[0,0].set_xlabel()

axs[0,1].plot() #apply to Axes [0,1]
```

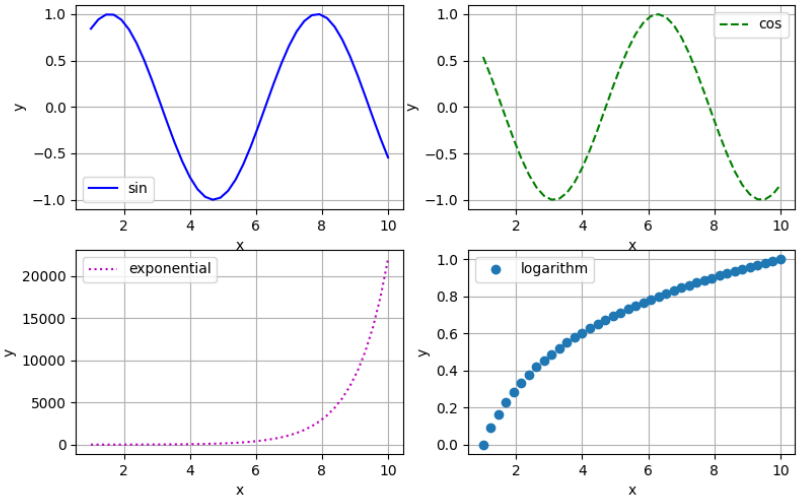



```
#make a row vector of 3 Axes Objects,
referenced by axs
fig, axs= plt.subplots(1, 3)
axs[0].bar() # apply to Axes [0]
axs[1].bar() # apply to Axes [1]
```



Subplot example

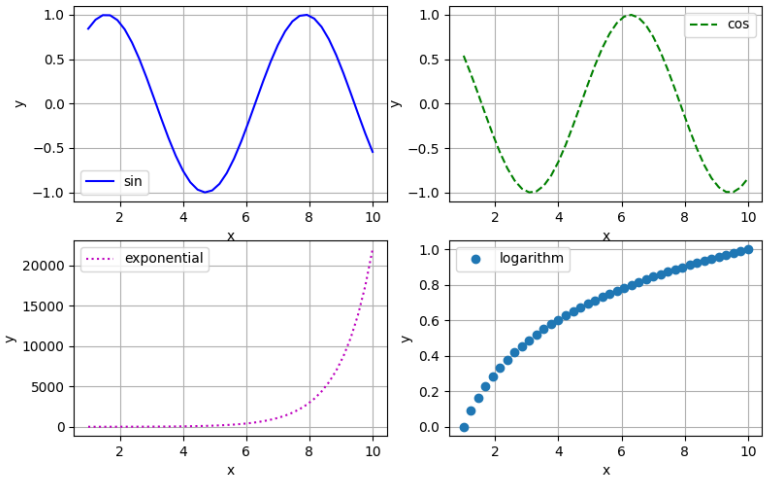

```
import matplotlib.pyplot as plt
import numpy as np
fig,axs =  plt.subplots(2,2, figsize=(12, 12))
x = np.linspace(1,10,num=40)

#Apply method to Axes element [0,0]
axs[0,0].plot(x,np.sin(x),'-b',label='sin')
axs[0,0].set_xlabel('x')
axs[0,0].set_ylabel('y')
axs[0,0].grid()
axs[0,0].legend()

#Apply method to Axes element [0.1]
axs[0,1].plot(x,np.cos(x),'--g', label='cos')
axs[0,1].set_xlabel('x')
axs[0,1].set_ylabel('y')
axs[0,1].grid()
axs[0,1].legend()

#Apply method to Axes element [1,0]
axs[1,0].plot(x,np.exp(x),':m',label='exponential')
axs[1,0].set_xlabel('x')
axs[1,0].set_ylabel('y')
axs[1,0].grid()
axs[1,0].legend()

#Apply method to Axes element [1,1]
axs[1,1].plot(x,np.log10(x),'o', label='logarithm')
axs[1,1].set_xlabel('x')
axs[1,1].set_ylabel('y')
axs[1,1].grid()
axs[1,1].legend()
plt.show()
```



Let's make the same subplots by using a for loop →

Use a loop to make subplots


```
import matplotlib.pyplot as plt
import numpy as np

fig,axs =  plt.subplots(2,2, figsize=(12, 12))
x = np.linspace(1,10,num=40)


#make lists that contain data we use within the loop
formats=['-b','g--', ':m','o'] # format strings for the plot
labels=['sin','cos','exponential','logarithm']
func=[np.sin,np.cos,np.exp,np.log10] # ufunc


for i,j in enumerate(axs.flatten()): # we flat the matrix of Axes, i is the
index and j is the Axes element
      j.plot(x,func[i](x),formats[i],label=labels[i])
      j.set_xlabel('x')
      j.set_ylabel('y')
      #j.set_xlim(1,10)
      #j.set_xticks(np.arange(1,10))
      j.grid()
      j.legend()

plt.show()
```



Create a custom plotting function


```
import matplotlib.pyplot as plt
import numpy as np
```

This function takes parameters: the Axes, etc, and returns the customized Axes

```
def myplot(ax0,xvalue,yvalue,formatstr, labelx, labely):
    ax0.plot(xvalue, yvalue,formatstr)
    ax0.set_xlabel(labelx)
    ax0.set_ylabel(labely)
    return ax0


x = np.linspace(1,10,num=40)
y=np.sin(x)


fig, ax = plt.subplots()
ax_out=myplot(ax,x,y,'--b', 'x','y') #call the function
plt.show()

fig, ax1 = plt.subplots()
y1=np.cos(x)
ax_out1=myplot(ax1,x,y1,'--r', 'x','y') # call the function
ax_out1.set_title("popolo") # you can apply methods to the returned
object
plt.show()
```

pandas

pandas (*panel data)* is a Python library designed for working with tabular data.

Built on NumPy's foundation, Pandas inherits and extends many of NumPy's array-based features.

Pandas is best used for working with heterogeneous and labeled data
NumPy is best used for working with homogeneous numerical arrays

If you do not have it, you can install pandas

```
conda install pandas  #if you have ananconda
pip3 install pandas   #if you do not have ananconda
```

To import Pandas

```
import pandas as pd
```

Pandas Object: Series

Characteristics and usage

- is a labeled 1D array - it is an analog of a 1D NumPy array with labeled indices.
- Homogeneous
- Mutable
- Fixed size
- Each element in a Series is associated with an index (label), which can be customized or automatically generated.

Usage of Series: Series are utilized to represent labeled data.
Example: storing student ages for an online course.
Series allows customization of indices (labels, depicted in blue), which means you can assign student names as labels, providing a more intuitive way to access each student's age directly by name.

```
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
```

Creating Series Objects


```
pd.Series(data=None, index=None)
```

parameters:

data: list, dictionary, ndarray, a scalar value
index: optional parameter, used to customize the indices.

Create a Series from a list

If the index parameter is not specified, labels are automatically generated to be integer numbers 0,1,2,.. like Python-style indices

```
Age= [38, 43, 38, 40, 49]
s=pd.Series(Age)
print(s)
0   38
1   43
2   38
3   40
4   49
dtype: int64
```

dtype: int64 is the dtype of the values

Create a Series from a list: customize the labels

index parameter: We can customize the indices by setting the index parameter to a list

```
Age = [38, 43, 38, 40, 49]
LastName = ["Sanchez", "Johnson", "Zhang", "Diaz", "Brown"]
sl=pd.Series(Age, index=LastName)
print(sl)
```

```
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
dtype: int64
```

We can also label the indices with nonsequential numbers:

```
sll=pd.Series(Age, index=[1,10,34,56,70])
print(sll)
```

```
1     38
10    43
34    38
56    40
70    49
dtype: int64
```

Create a Series from a dictionary

A Series can be created out of a dictionary,
in which case the indices default to the dictionary keys:

```
D={'Sanchez': 38, 'Johnson': 43, 'Zhang': 38,
   'Diaz': 40, 'Brown': 49}

sd=pd.Series(D)
print(sd)
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
dtype: int64
```

A series can also be seen as dictionary-like, where each value has an associated label

Create a Series from a dictionary: index parameter

index parameter: In the case of a dictionary, the index parameter can be explicitly set to control the order and/or the subset of keys used.

```
D={'Sanchez': 38, 'Johnson': 43, 'Zhang': 38, 'Diaz': 40, 'Brown': 49}
L=['Brown', 'Diaz', 'Johnson', 'Sanchez', 'Zhang'] # list of keys sorted
```

If we set the index to L, the order of the elements in the Series follows the elements in the list:

```
sd=pd.Series(D, index =L)
print(sd)
```

```
Brown      49
Diaz       40
Johnson    43
Sanchez    38
Zhang      38
dtype: int64
```

We can also make a Series from a subset of key: value pairs in the order we decide:

```
sd=pd.Series(D, index =['Diaz', 'Johnson'])
print(sd)
```

```
Diaz       40
Johnson    43
dtype: int64
```

Attributes of a Series

A Series is like a 1D array, and it has similar attributes:

```
print(s)
0   38
1   43
2   38
3   40
4   49
dtype: int64

s.ndim  #1
s.shape #(5,)
s.size  #5

s.values #returns a 1D array of the values
[38 43 38 40 49]

s.index  #returns the Index Object of the labels
RangeIndex(start=0, stop=5, step=1)
```

means a range of integers in range [0 ,5) with a step of 1.

```
s.index.values #returns the labels as 1D array
[0 1 2 3 4]
```

Attributes of a Series

```
print(sd)
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
dtype: int64
```

```
sd.index
Index(['Sanchez', 'Johnson', 'Zhang', 'Diaz', 'Brown'],
dtype='object')
```

dtype='object' specifies the data type of the elements. Here, 'object' typically denotes strings in Pandas.

```
sd.index.values
['Sanchez' 'Johnson' 'Zhang' 'Diaz' 'Brown'] #1D array of strings.
```

In pandas the type of the labels is an Index Object

Series Type Conversion

Converting a Series to a 1D NumPy array

```
np.array(sd)
sd.values)
sd.to_numpy()
[38 43 38 40 49]
```

```
print(sd)
Sanchez    38
Johnson    43
Zhang      38
Diaz       40
Brown      49
dtype: int64
```

Convert a Series to a list type

```
list(sd)
sd.to_list()
[38, 43, 38, 40, 49]
```

Converting a Series to a dictionary type

```
dict(zip(sd.index, sd.values))
sd.to_dict()
{'Sanchez': 38, 'Johnson': 43, 'Zhang': 38, 'Diaz': 40, 'Brown': 49}
```

DataFrame Object

Characteristics and usage

- is a 2D labeled tabular structure, and it is an analog of a 2D NumPy array with labelled rows and columns (depicted in blue)
- Heterogeneous
- Mutable
- Size can change
- is a collection of Series
- each element in a DataFrame is associated with row and column indices (labels), which can be customized or automatically generated.

Usage: DataFrames are ideal for handling heterogeneous data with labeled rows and columns and for representing tabular data: rows correspond to instances (examples, observations, etc.), and columns correspond to features of these instances.

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

Creating DataFrame Objects

To create a DataFrame out of other python Objects, we will use the pd.DataFrame() constructor.

```
pd.DataFrame(data=None, index=None, columns=None)
```

data: dictionary can contain ndarray, lists, Series
- 2D ndarray
- Series
- Pandas DataFrame

index: sets the row labels. Default row indices are 0,1,2..
columns: sets the column labels. The default column indices are 0,1,2..

Create a DataFrame from a 2D array


```
arr1=np.array([[1, 1, 1], [2, 4, 8], [3, 9, 27], [4, 16, 64], [5, 25, 125]])
dfnp = pd.DataFrame(arr1) #if index and column parameters are missing, indices automatically default to python-style 0,1,2,
print(dfnp)
```

	0	1	2
0	1	1	1
1	2	4	8
2	3	9	27
3	4	16	64
4	5	25	125

To customize the row and column labels we set the index and columns parameters.

```
index_val=['first','second','third','fourth','fifth'] #list of row labels
column_val=['number', 'squares', 'cubes']   #list of column labels
dfnp = pd.DataFrame(arr1, index=ndex_val, columns=column_val)
print(dfnp)
```

	number	squares	cubes
first	1	1	1
second	2	4	8
third	3	9	27
fourth	4	16	64
fifth	5	25	125

Create a DataFrame from a dictionary of lists


```
D={"LastName": ["Sanchez", "Johnson", "Zhang", "Diaz", "Brown"],
"Age": [38, 43, 38, 40, 49], "Height": [71.2, 69.0, 64.5, 67.4, 64.2],
"Weight": [176.1, 163.5, 131.6, 133.1, 119.8]}

df=pd.DataFrame(D)
print(df)
```

	LastName	Age	Height	Weight
0	Sanchez	38	71.2	176.1
1	Johnson	43	69.0	163.5
2	Zhang	38	64.5	131.6
3	Diaz	40	67.4	133.1
4	Brown	49	64.2	119.8

The dictionary keys will be used as column labels and the values in each list as the values (data) in the columns of the DataFrame.

If we do not use the index parameter, pandas automatically generates the row labels, which default to the the normal Python indices 0,1,2,..

Create a DataFrame from a dictionary of lists – customize row labels

We can **customize the row** labels by defining them via the **index parameter** of the pd.DataFrame()

```
 D={"LastName": ["Sanchez", "Johnson", "Zhang", "Diaz", "Brown"],
"Age": [38, 43, 38, 40, 49], "Height": [71.2, 69.0, 64.5, 67.4, 64.2],
"Weight": [176.1, 163.5, 131.6, 133.1, 119.8]}

dfc=pd.DataFrame(D, index=['A','B','C','D','E'])
print(dfc)
```

```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

Changing row labels and column labels of an existing DataFrame

We can change row and column labels using attributes:

```
df.columns = new_columns
df.index = new_index
```

```
print(dfnp)
```

	number	squares	cubes
first	1	1	1
second	2	4	8
third	3	9	27
fourth	4	16	64
fifth	5	25	125

To change the row labels:

```
dfnp.index=[10,20,30,40,50]
print(dfnp)
```

	number	squares	cubes
10	1	1	1
20	2	4	8
30	3	9	27
40	4	16	64
50	5	25	125

To change the column labels:

```
dfnp.columns=['A', 'B','C']
print(dfnp)
```

	A	B	C
10	1	1	1
20	2	4	8
30	3	9	27
40	4	16	64
50	5	25	125

Changing row labels of an existing DataFrame

The set_index() method is used to set the row labels using existing columns

```
print(dfnp)
```

```
   A   B    C
10  1   1    1
20  2   4    8
30  3   9   27
40  4  16   64
50  5  25  125
```

```
df1=dfnp.set_index('C') #we set the existing column "C" as row labels
```

```
     A   B
C
1    1   1
8    2   4
27   3   9
64   4  16
125  5  25
```

The set_index() method returns a new object which is a copy of the original DataFrame object. If you want the original DataFrame object to be modified, you can use the parameter **inplace=True**.

```
dfnp.set_index('B', inplace=True)  #will modify directly dfnp
```

DataFrame - the info() method

The info() method in Pandas provides a concise summary of a DataFrame, including information about the index, columns, data types, non-null values, and memory usage.

print(df)

```
  LastName  Age  Height  Weight
0  Sanchez   38    71.2   176.1
1  Johnson   43    69.0   163.5
2    Zhang   38    64.5   131.6
3     Diaz   40    67.4   133.1
4    Brown   49    64.2   119.8
```

print(df.info())

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   LastName  5 non-null      object
 1   Age       5 non-null      int64
 2   Height    5 non-null      float64
 3   Weight    5 non-null      float64
dtypes: float64(2), int64(1), object(1)
memory usage: 288.0+ bytes
None
```

DataFrame - attributes

NumPy-like attributes

```
df.size
df.shape
df.ndim
```

Additional attributes

```
df.index          #returns Index Object of row labels
df.columns        #returns Index Object of column labels

df.index.values    #returns 1D array of row labels
df.columns.values #returns 1D array of column labels


df.index.dtype      #returns the type of row labels
df.columns.dtype    #return the type of the column labels
```

Data Indexing and Selection

Indexers: loc [] and iloc[]

An indexer is a mechanism that allows users to locate and access specific subsets of data, including rows, columns, or individual elements.

Pandas main two Indexers are:

.loc[] allows indexing methods that always reference the labels of rows and columns, the visible ones, no matter if they are the defaults (0,1,2, etc) of customized.
Use: convenient when you want to select data using row and column labels.

.iloc[] allows indexing methods that always reference the Python-style indices based on position (0,1,2, .. no matter if they are visible or not).
Use: convenient choice when labels are not significant or when you want to perform operations based on the numerical position of data. This is usually faster than loc, and maintains consistency with the use integer-based indexing of NumPy arrays.

The use of loc[] and iloc[] can prevent subtle bugs due to the mixed indexing/slicing convention. Alo iloc

Indexing a Series

A Series can be indexed by using loc[] or iloc[]:

```
.iloc[] #use positional indices – usual python/NumPy indexing syntax

iloc[index]                          Single element Indexing
iloc[[index1,index2, index3]]        Array Indexing (Fancy indexing)
iloc[ index_start : index_end]       Slicing, the start is included, end excluded
iloc[boolean array]                  Boolean Indexing (Masking)
```

```
.loc[] #use the labels (defaults or customized)

loc[label]                           Single element Indexing
loc[[ label1, label2, label3 ]]      Array Indexing (Fancy indexing)
loc[ label_start: label_end ]        Slicing - both the start and end included
loc[boolean array/series]            Boolean Indexing (Masking)
```

Indexing a Series – iloc[ ]

When using iloc[] we index a Series as you would index a 1D array (no matter what the labels are)

```
print(sl)
Sanchez     38
Johnson     43
Zhang       38
Diaz        40
Brown       49
dtype: int64
```

```
print(sl.iloc[1])  #single element indexing
43

print(sl.iloc[ [1,3]]) #array indexing
Johnson     43
Diaz        40
dtype: int64
```

To use Boolean indexing with iloc[], we convert the Boolean Series to a Boolean array. Iloc[] does not support a Boolean series, while loc[] does.

```
bool1= sl > 30) & (sl < 50) #creates a Boolean Series
arr1= np.array(bool1)
print(sl.iloc[ arr1] )
Johnson     43
Brown       49
dtype: int64
```

Indexing a Series – iloc[ ]

To index a Series with iloc[] we use the same Python syntax we would use to index a 1D array (no matter what the labels are). iloc[]. uses positional indices.

```
print(s)
0   38
1   43
2   38
3   40
4   49
dtype: int64
```

```
s.iloc[0] #access first element
38
```

```
s.iloc[[1,3]] #indexing with a list of indices
1    43
3    40
dtype: int64
```

```
s.iloc[:2] #slice from index 0 to index 1
0    38
1    43
dtype: int64
```

Indexing a Series – loc[ ]

To index a Series with loc[] we use the labels.
To obtain a 1D array of the labels:

```
print(sl)
Sanchez     38
Johnson     43
Zhang       38
Diaz        40
Brown       49
dtype: int64
```

```
print(sl.index.values)
['Sanchez' 'Johnson' 'Zhang' 'Diaz' 'Brown']
```

```
sl.loc['Sanchez'] #access the value labeled with 'Sanchez'
38
```

```
sl.loc[:'Zhang']  #slice to the element labeled with 'Zhang'
Sanchez     38
Johnson     43
Zhang       38
dtype: int64
```

```
sl.loc[['Sanchez','Zhang']] #select multiple elements with list of labels
Sanchez     38
Zhang       38
dtype: int64
```

```
bool2=(sl > 40) & (sl < 50) #masking with Boolean Series
sl.loc[bool2]
Johnson     43
Brown       49
dtype: int64
```

Indexing a Series – loc[ ]

Now we use loc[] on this Series, where the indices default to integer numbers 0, 1, 2, .. .
They are the row labels in this case.

To obtain a 1D array of row labels:

```
print(s.index.values)
[0 1 2 3 4]
```

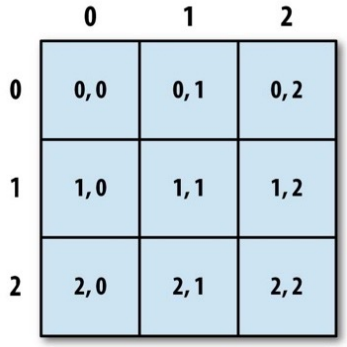
```
print(s)
0   38
1   43
2   38
3   40
4   49
dtype: int64
```

Now we use those labels to index with loc[]

```
print(s.loc[:2]) #notice when slicing with loc the end label is inlcuded
0    38
1    43
2    38
dtype: int64dtype: int64
```

Indexing a DataFrame with iloc[]

.iloc[] integer-Based Indexing (like indexing a 2D array): with .iloc, you can select rows and columns based solely on their integer positions, regardless of the row and column labels



Single Label Indexing:

df.iloc[row_index] Accesses the row at index row_index.

df.iloc[:, column_index] Accesses the column at index column_index.

Slicing with Indices:

df.iloc[start_index :end_index] Accesses rows from start_index to end_index-1.

df.iloc[:, start_index:end_index] Accesses columns from start_index to end_index-

Array Indexing with Indices:

df.iloc[[index1, index2, index3]] Accesses rows at indices index1, index2, and index3.

df.iloc[:, [index1, index2, index3]] Accesses columns at indices index1, index2, and index3.

Combination of Indices and Slicing:

df.iloc[row_index, column_index] Accesses the element at row row_index, column column_index.

df.iloc[start_row:end_row, start_column:end_column] Accesses a subset of rows and columns.

Indexing a DataFrame with iloc[]

Use same syntax you would use to index a 2D array

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

df.iloc[::2] # slice every two rows

LastName	Age	Height	Weight
A Sanchez	38	71.2	176.1
C Zhang	38	64.5	131.6
E Brown	49	64.2	119.8

df.iloc[[1,3]] # list of indices [1,3] to select 2nd and 4th row

LastName	Age	Height	Weight
B Johnson	43	69.0	163.5
D Diaz	40	67.4	133.1

df.iloc[1,3] # select one element at row index 1, column index 3

163.5

Indexing a DataFrame with iloc[]

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

```
df.iloc[2] # same as df.iloc[2,:] select 3rd row, returns a series
LastName    Zhang
Age            38
Height       64.5
Weight      131.6
Name: 2, dtype: object
```

```
df.iloc[[2]] # returns a dataframe
```

	LastName	Age	Height	Weight
C	Zhang	38	64.5	131.6

Indexing a DataFrame with iloc[]

df.iloc[:,3] # select 4th column

```
A    176.1
B    163.5
C    131.6
D    133.1
E    119.8
Name: Weight, dtype: float64
```

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

df.iloc[:,[1,3]] # select 2nd and 4th columns

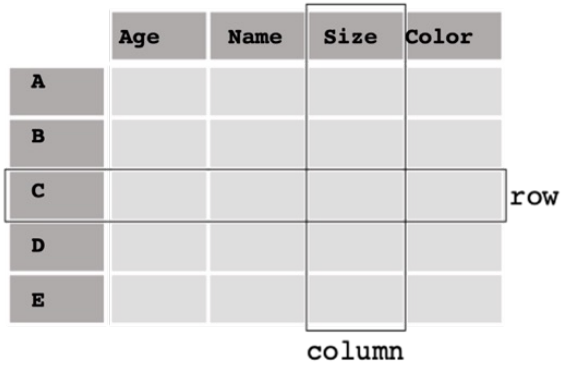
	Age	Weight
A	38	176.1
B	43	163.5
C	38	131.6
D	40	133.1
E	49	119.8

df.iloc[[1,3],[1,3]] # select 2nd and 4th rows, 2nd and 4th columns

	Age	Weight
A	43	163.5
D	40	133.1

Indexing a DataFrame with loc[ ]

The loc[] provides **label-based indexing**, When using .loc, you can specify rows and columns based on their labels, regardless of whether they are the default integer indices or custom index labels.



Single Label Indexing:

df.loc[row_labe'] Accesses the row with label row_label.

df.loc[:, column_label] Accesses the column with label column_label.

Slicing with Labels:

df.loc[start_label : end_label] Accesses rows from start_label to end_label (inclusive).

df.loc[:, start_label : end_label] Accesses columns from start_label to end_label (inclusive).

Array Indexing:

df.loc[[label1, label2, label3]] Accesses rows with labels label1, label2, and label3.

df.loc[:, ['label1, label2, label3]] Accesses columns with labels label1, label2, and label3

Combination of Labels and Slicing:

df.loc[row_label, column_label] Accesses the element at row row_label, column column_label.

df.loc[start_row : end_row, start_column : end_column] Accesses a subset of rows and columns.

Indexing a DataFrame with loc[ ]

	LastName	Age	Height	Weight
A	Sanchez	38	71.2	176.1
B	Johnson	43	69.0	163.5
C	Zhang	38	64.5	131.6
D	Diaz	40	67.4	133.1
E	Brown	49	64.2	119.8

When using loc[] use the labels

```
df.columns.values #return 1D array of the row labels
['LastName', 'Age', 'Height', 'Weight']

df.index,values #return 1D array of the column labels
['A', 'B', 'C', 'D', 'E']
```

```
df.loc['A','Height'] # select one element at row label 'A' and
columns label 'Height'
```

Indexing a DataFrame with loc[ ]


```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

```
dfc.loc['A'] # selects row labeled 'A', and returns it as Series
LastName    Sanchez
Age              38
Height         71.2
Weight        176.1
Name: A, dtype: object
```

If you want to select and return a row as a DataFrame, you should pass a list:

```
dfc.loc[['A']] #a list of labels returns a DataFrame
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
```

Indexing a DataFrame with loc[ ]

Use labels

```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

df.loc[:,'Age':] # select from column 'Age' to the end

```
   Age  Height  Weight
A   38    71.2   176.1
B   43    69.0   163.5
C   38    64.5   131.6
D   40    67.4   133.1
E   49    64.2   119.8
```

df.loc[['A','D'],'Age':] #select rows 'A' and 'D' and columns 'Age' to the end

```
   Age  Height  Weight
A   38    71.2   176.1
D   40    67.4   133.1
```

Indexing a DataFrame with loc[ ]


```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

```
df.loc[['D','A'],['Age','Height']]
```

```
    Age  Height
D    40    67.4
A    38    71.2
```

```
df.loc['D':] # select rows from 'D' to the end
```

```
df.loc['D'] # select row D
```

Other ways to select columns: use [ ] or dot notation


```
  LastName  Age  Height  Weight
A  Sanchez   38    71.2   176.1
B  Johnson   43    69.0   163.5
C    Zhang   38    64.5   131.6
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

You can use [ ]

```
df['Age'] # select column 'Age' and returns a series
```

```
df[['Age']] # select column 'Age' and returns a dataframe
```

```
df[['LastName','Height']] # select columns 'LastName' and 'Height'
```

Or you can use the dot notation, also called attribute access

```
df.Age
0    38
1    43
2    38
3    40
4    49
Name: Age, dtype: int64
```

Boolean Indexing

We extract information based on conditions.

```
comparison operators:
==  equals
!= not equals
> greater than
< less than
>= greater than or equal to
<= less than or equal to

Logical operator - each condition must be put in a
separate pair of brackets.
  & (and)
  | (or
  ~ (not)
```

Boolean Indexing with loc[]

loc[] works with label and also with Boolean Series, Boolean array, and a list of Boolean values.

The index boolean can be: Boolean Series, a list of Boolean, and ndarray of Boolean

Boolean Indexing with Rows:

df.loc[boolean] Selects rows where the corresponding value in boolean is True.

Boolean Indexing with Columns:

df.loc[:, boolean] Selects columns where the corresponding value in Boolean is True.

Combining Boolean Indexing:

df.loc[boolean, column_label] Selects True rows and the column specified by column_label

df.loc[boolean, [label1, label2, label3]] Selects True rows and specific columns

df.loc[boolean, start_column : end_column] Selects True rows and columns from start_column to end_column (inclusive).

Boolean Indexing – use loc[]


```
#notice each condition is between parenthesis
bool2=(dfc['Age'] >= 40) & (dfc['Height'] < 70)

df.loc[bool2] # extract rows satisfying the condition
```

```
  LastName  Age  Height  Weight
B  Johnson   43    69.0   163.5
D     Diaz   40    67.4   133.1
E    Brown   49    64.2   119.8
```

```
df.loc[bool2,'LastName'] # extract LastName column of the rows
satisfying the condition
```

```
B    Johnson
D       Diaz
E      Brown
Name: LastName, dtype: object
```

```
df.loc[bool2,['Age','LastName']]
```

```
   Age LastName
B   43  Johnson
D   40     Diaz
E   49    Brown
```

Boolean Indexing – use iloc[]

iloc[] works with integer indices (like NumPy) and with a Boolean array or a list of Boolean values. Does not work with Boolean Series.

Boolean Indexing with Rows:
df.iloc[boolean] Selects rows where the corresponding value in boolean is True.

Boolean Indexing with Columns:
df.iloc[:, boolean] Selects columns where the corresponding value in boolean is True.

Combining Boolean Indexing:
df.iloc[boolean, column_index] Selects True rows and the column at column_index.

df.iloc[boolean, start_column:end_column] Selects True rows and columns from start_column to end_column (inclusive).

df.loc[boolean, [column1, column2, colum3]] Selects True rows specific columns

If you use iloc[] you should convert the Boolean Series to a numpy array or list
`df.iloc[np.array(bool1)])`

Read in data and store it in a DataFrame

```
read_csv(filename, sep, index_col, names, header, skirows, parse_dates)
```

sep='character', defines the field separator. By default, is set to ','
sep='\s+' will set the field separator to whitespaces, including tabs

Select a field to label the rows
index_col=field number (starts from 0).
If not specified, rows are indexed with integer numbers 0 to n-1.

Specify column labels
names=list of values If not specified, column labels are taken from the first row of the file

Specify which row to be used to label the columns
header=0, the first row is considered as the header.
header = None will assign default integer numbers to the column labels.
header=number - will pick a row number to label the columns

Skip comment lines
comment='character'

Read in data and store it in a DataFrame

```
read_csv(filename, sep, index_col, names, header, skirows, parse_dates)
```

Skip comment lines
comment='character’

Skip rows
skiprows=int - will skip int number of lines
skiprows=list of numbers - will skip these row numbers

Read in specific fields
usecols=list of numbers, or list of names - to read in specific columns

Deadline with datetime data
parse_dates convert the specified columns, containing date or datetime-like strings, into datetime objects.
parse_dates=['Column1', 'Column2', ...]

Read in data and store it in a DataFrame - Examples

read_csv() uses a comma as the default separator.

sample-csv.csv

```
name,age,state,point
Alice,24,NY,64
Bob,42,CA,92
Charlie,18,CA,70
Dave,68,TX,70
Ellen,24,CA,88
Frank,30,NY,57
```

Here we just read the data set, without using other parameters

```
# column names are taken from the first row of the file
# row indices are set to 0 .. n-1
df1=pd.read_csv('sample-csv.csv')
```

```
      name  age state  point
0    Alice   24    NY     64
1      Bob   42    CA     92
2  Charlie   18    CA     70
3     Dave   68    TX     70
4    Ellen   24    CA     88
5    Frank   30    NY     57
```

Read in data and store it in a DataFrame - Examples

read_csv() uses a comma as the default separator.
sample-csv.csv

```
name,age,state,point
Alice,24,NY,64
Bob,42,CA,92
Charlie,18,CA,70
Dave,68,TX,70
Ellen,24,CA,88
Frank,30,NY,57
```

To label the rows with a specific field:

```
#index_col=0 will label the rows with the first field
df2=pd.read_csv('sample-csv.csv', index_col=0)
```

name	age	state	point
Alice	24	NY	64
Bob	42	CA	92
Charlie	18	CA	70
Dave	68	TX	70
Ellen	24	CA	88
Frank	30	NY	57

Read in data and store it in a DataFrame - Examples

To label the columns with a specific row:

```
# Label the columns with the second row
df3=pd.read_csv('sample-csv.csv', header=2)
```

To read in specific fields, and so skip others:

```
#read in 3rd and 4th fields
df4=pd.read_csv('sample-csv.csv', usecols=[2,3])
```

Read in data and store it in a DataFrame - Examples

sample-comments-sep.txt

```
#comment lines
#comment lines
name:age:state:point
Alice:24:NY:64
Bob:42:CA:92
Charlie:18:CA:70
Dave:68:TX:70
#comment lines
Ellen:24:CA:88
Frank:30:NY:57
```

Set the 3rd field to label the rows, and exclude comment lines

```
df5=pd.read_csv('sample-comments-sep.txt', sep=':',
       comment='#', index_col=2)
```

state	name	age	point
NY	Alice	24	64
CA	Bob	42	92
CA	Charlie	18	70
TX	Dave	68	70
CA	Ellen	24	88
NY	Frank	30	57

Read in data and store it in a DataFrame - Examples

Here is an example of a data set containing dates, sample-dates.csv

```
Name,Age,State,Score,Birthdate
Alice,24,NY,64,1999-05-15
Bob,42,CA,92,1981-02-28
Charlie,18,CA,70,2006-11-03
Dave,68,TX,70,1956-07-20
Ellen,24,CA,88,1999-08-12
Frank,30,NY,57,1993-10-05
```

```
df5=pd.read_csv(sample-date.csv, parse_dates=['Birthdate'])
```

Converting to a datetime is very useful when you want to plot the datetime values.
You can also explore the pd.to_datetime() function for converting fields to a datetime object.

To read in data pandas provides other functions to read excel, jason files etc.
More info here https://pandas.pydata.org/docs/user_guide/io.html

DataFrame operations useful for data analysis

Getting info about the data

info() provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.

```
df.info()
```

Viewing the data

head() and tail() methods are bash-like commands.

By default, they output the first and last five rows of a DataFrame, but we could also pass a number.

```
df.head(10) #outputs the top ten rows
```

DataFrame operations useful for data analysis

Check and remove duplicate rows

Check for duplicate rows

```
df.duplicated() #returns a Boolean Series
```

To know how many duplicate roes there are we use the sum function, which will sum the True values:

```
df.duplicated().sum() #sum the True values
```

Remove duplicate rows

```
df.drop_duplicates(inplace=True)  #drops duplicate rows
```

DataFrame operations useful for data analysis

Detect and remove missing values

Missing values are represented in padas as NaN for numeric and string values, and with NaT for datetime values.

We use **isnull() or isna()** for detecting missing values:

```
df.isnull() #returns a Boolean DataFrame

df.isnull().sum() #total number of missing values (True) in each
                  column
```

You can remove missing values by using **dropna()**

```
df.dropna(inplace=True) #delete any row containing missing values
```

#you can also drop columns containing missing values by setting axis=1

```
df.dropna(axis=1, inplace=True)
```

Adding rows

To add a row in DataFrame, we can use the concat() function, which concatenates DataFarmes. This function is useful if you want to concatenate different data sets.

	LastName	Age	Height	Weight
0	Sanchez	38	71.2	176.1
1	Johnson	43	69.0	163.5
2	Zhang	38	64.5	131.6
3	Diaz	40	67.4	133.1
4	Brown	49	64.2	119.8

Create a DataFrame containing rows:

```
new_row = pd.DataFrame.from_dict({'LastName': ['Clara'],
                                  'Age': 40,
                                  'Height': 70.0})
```

Concatenate the two DataFrames

```
df = pd.concat([new_row, df], ignore_index=True)
```

	LastName	Age	Height	Weight
0	Sanchez	38	71.2	176.1
1	Johnson	43	69.0	163.5
2	Zhang	38	64.5	131.6
3	Diaz	40	67.4	133.1
4	Brown	49	64.2	119.8
5	**Clara**	**40**	**70.0**	**NaN**

ignore_index=True is used to re-set the indices of the resulting dataframe

Add a column at the end using [ ]

	LastName	Age	Height	Weight
0	Sanchez	38	71.2	176.1
1	Johnson	43	69.0	163.5
2	Zhang	38	64.5	131.6
3	Diaz	40	67.4	133.1
4	Brown	49	64.2	119.8

We want to add the High Blood Pressure values of patients at the end the DataFrame df.

```
L=[124, 109, 125, 117, 122] # define a list of values
```

Add the list as a column, and name the column 'HighBP'

```
df["HighBP"]=L
```

	LastName	Age	Height	Weight	HighBP
0	Sanchez	38	71.2	176.1	124
1	Johnson	43	69.0	163.5	109
2	Zhang	38	64.5	131.6	125
3	Diaz	40	67.4	133.1	117
4	Brown	49	64.2	119.8	122

Add a column at the end using [ ]

You can also perform vectorized operations between columns (which are series type) , and add the result to a DataFrame

```
df["BMI"]=(df['Weight']*0.453592)/(df['Height']*0.0254)**2
```

	LastName	Age	Height	Weight	HighBP	BMI
0	Sanchez	38	71.2	176.1	124	24.422905
1	Johnson	43	69.0	163.5	109	24.144462
2	Zhang	38	64.5	131.6	125	22.239981
3	Diaz	40	67.4	133.1	117	20.599478
4	Brown	49	64.2	119.8	122	20.435474

Add a column at a specific position using insert()

Dataframe.insert() is used to insert a column to a Dataframe at a specified index position. It is like the list insert method, and it updates the original DataFrame.

The general syntax is:

```
df.insert(index_position, column_name, value)
```

For example, we want to add Low Blood Pressure values after the HighBP column. We want to name the column LowBP, and insert the column at index 5, which is the 6th column.

```
Lv=[60, 75, 67, 85, 90, 82] #define a list of values
df.insert(5,'LowBP',Lv)
```

	LastName	Age	Height	Weight	HighBP	LowBP	BMI
0	Sanchez	38	71.2	176.1	124	60	24.422905
1	Johnson	43	69.0	163.5	109	75	24.144462
2	Zhang	38	64.5	131.6	125	67	22.239981
3	Diaz	40	67.4	133.1	117	85	20.599478
4	Brown	49	64.2	119.8	122	90	20.435474

Remove rows and columns - drop() method

To delete columns and rows of DataFrame, we can use the **drop()** method. The drop() method by default returns a new DataFrame with the modified values. If you want to modify the same dataFrame, use inplace=True

	LastName	Age	Height	Weight
0	Sanchez	38	71.2	176.1
1	Johnson	43	69.0	163.5
2	Zhang	38	64.5	131.6
3	Diaz	40	67.4	133.1
4	Brown	49	64.2	119.8

To delete columns, use column labels, and axis=1

```
df.drop(['Height','Weight'],axis=1) #delete columns
```

To delete rows, use row labels, and axis=0, which is default

```
df.drop([0,3]) #delete rows
```

You can also use the inplace=True

Sorting

To sort you can use:

- sort_values() to sort values (the data) along an axis
- sort_index() to sort labels along an axis

axis=0 (default) row-wise, axis=1 column-wise

	LastName	Age	Height	Weight	BMI
4	Brown	49	64.2	119.8	20.435474
1	Johnson	43	69.0	163.5	24.144462
3	Diaz	40	67.4	133.1	20.599478
0	Sanchez	38	71.2	176.1	24.422905
2	Zhang	38	64.5	131.6	22.239981

We want to sort values row-wise by a column. Default is ascending order.

```
df.sort_values('Age', ascending=False) # sort values by Age
```

	LastName	Age	Height	Weight	BMI
4	Brown	49	64.2	119.8	20.435474
1	Johnson	43	69.0	163.5	24.144462
3	Diaz	40	67.4	133.1	20.599478
0	Sanchez	38	71.2	176.1	24.422905
2	Zhang	38	64.5	131.6	22.239981

Explore the parameters ignore_index of the sort_values().

Sorting

To sort you can use:

- sort_values() to sort values (the data) along an axis
- sort_index() to sort labels along an axis

axis=0 (default) row-wise, axis=1 column-wise

To sort by row labels:

```
df.sort_index())
```

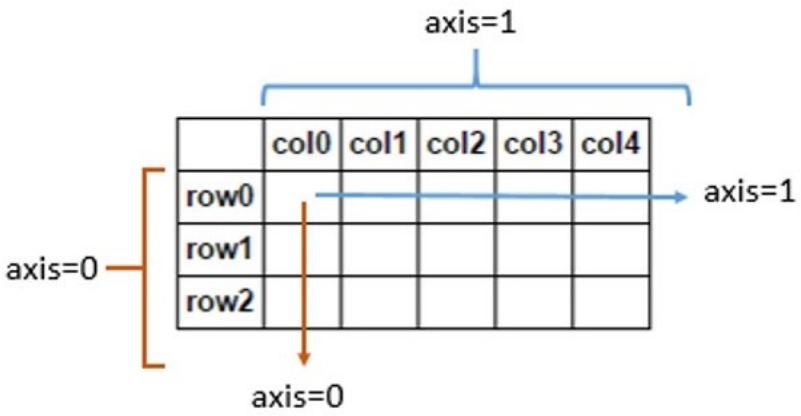
LastName	Age	Height	Weight	BMI
Sanchez	38	71.2	176.1	24.422905
Johnson	43	69.0	163.5	24.144462
Zhang	38	64.5	131.6	22.239981
Diaz	40	67.4	133.1	20.599478
Brown	49	64.2	119.8	20.435474

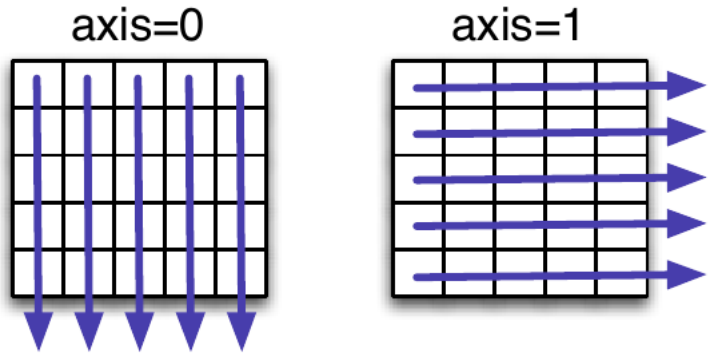
To sort by column labels:

```
df.sort_index(axis=1))
```

LastName	Age	BMI	Height	Weight
Sanchez	38	24.422905	71.2	176.1
Johnson	43	24.144462	69.0	163.5
Zhang	38	22.239981	64.5	131.6
Diaz	40	20.599478	67.4	133.1
Brown	49	20.435474	64.2	119.8

Statistics methods





axis=0 (default) means operations are performed row-wise, i.e., along the vertical axis.
axis=1 means operations are performed column-wise, i.e., along the horizontal axis.

```
count()   - number of non-NA observations
sum()     - sum of values
mean()    - mean of values
min()     - minimum
max()     - maximum
abs()     - absolute Value
prod()    - product of values
std()     - standard deviation
cumsum()  - cumulative sum
cumprod()- cumulative product
idxmin()  - index of the minimum
idxmax()  - index od the maximum
```

Statistics methods - Examples

```
m=df['Age'].min() #minimum of a Series

idm=df['Age'].idxmin() # index of the minimum of a Series


mv=df.min(axis=0) #minimum down (vertically)


mh=df.min(axis=1) #minimum across (horizontally)
```

Generate summary of statistics

The **describe()** method is very useful because returns a summary statistics of a DataFrame for each column.

	LastName	Age	Height	Weight	BMI
0	Sanchez	38	71.2	176.1	24.422905
1	Johnson	43	69.0	163.5	24.144462
2	Zhang	38	64.5	131.6	22.239981
3	Diaz	40	67.4	133.1	20.599478
4	Brown	49	64.2	119.8	20.435474

```
df.describe() #returns a DataFrame
```

	Age	Height	Weight	BMI
count	5.000000	5.000000	5.000000	5.000000
mean	41.600000	67.260000	144.820000	22.368460
std	4.615192	2.981275	23.798676	1.887933
min	38.000000	64.200000	119.800000	20.435474
25%	38.000000	64.500000	131.600000	20.599478
50%	40.000000	67.400000	133.100000	22.239981
75%	43.000000	69.000000	163.500000	24.144462
max	49.000000	71.200000	176.100000	24.422905

Writing data

You can write data in csv format by using the **to_csv** function
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html

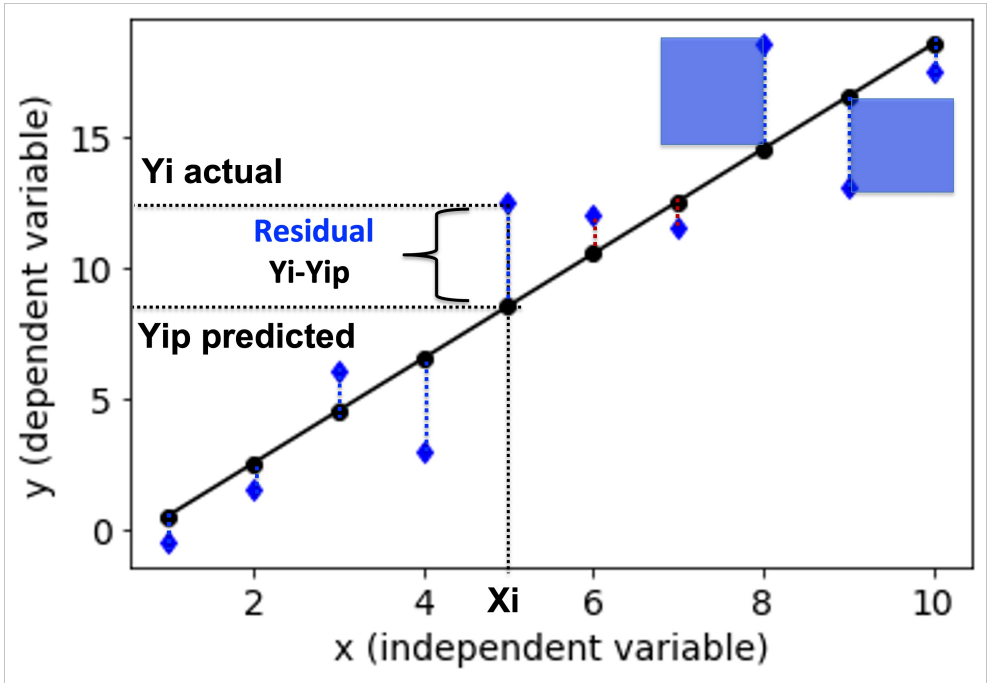
Looping

```
for col in df:  #loop over column labels
     print(col)


for r in df.index.values:  #loop over row labels
     print(r)
```

You can also use df.**iterrows()** to iterate over DataFrame rows as (index, Series) pairs.

Least-Squares method

The Least Square method is the process of finding a regression line or best-fitted line for the observed data by minimizing **the sum of the squares of the residuals** between the data the fitting model



Sum of squared residuals (SSR) is basically the sum the blue squares in the above figure. Mathematically it is described as:

$$SSR = \sum_i (y_i - y_{ip})^2$$

Linear models - Coefficient of determination (R^2)

The coefficient of determination **R^2** is popularly used to determine the quality of the fit. It is defined as:

$$R^2 = 1 - \frac{\sum_i (y_i - y_{ip})^2}{\sum_i (y_i - \bar{y})^2}$$

Where:
y_i – the i[th] y data
y_{ip} – the i[th] predicted y value from the model
$\bar{y}$ – the average of the y data

If R^2 close to 1 the model is a good approximation to the data compared to the mean of the data, but be careful not to overfit data

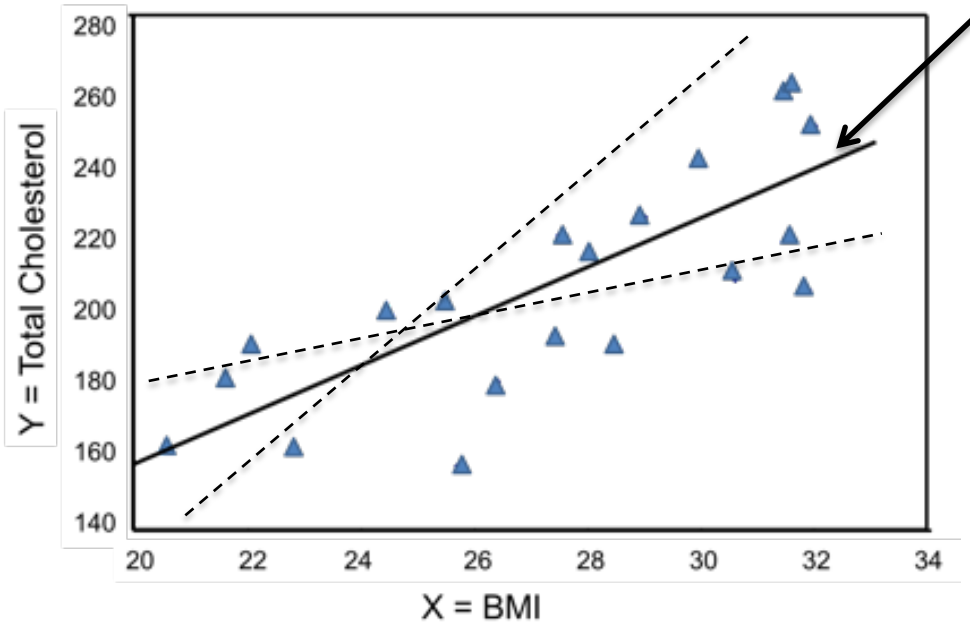
If R^2 close to 0 the model is poor, as the two deviations will be comparable

Example

Below is an example of a linear regression analysis, showing the dependence of the total cholesterol (dependent variable) on the Y-axis vs body mass index (independent variable) on the X-axis:

Model: Linear equation:

$y = a+b\,x$ (a and b are the fitting parameters)



Fitted Model

$y = 28.07 + 6.49\,x$

$R^2 = 0.67$

Fitted parameters

a=28.07

b=6.49

Least Square Regression method finds the line that minimizes the differences between observed and predicted values of the outcome.

Linear models

Linear models are linear functions, i.e., linear in their parameters, β0, β1 etc.

Polynomials are linear models

degree 1 – linear function $y = \beta 0\, x + \beta 1$

degree 2 – quadratic function $y = \beta 0\ x^2 + \beta 1\ x + \beta 2$

degree 3 – cubic function $y = \beta 0\, x^3 + \beta 1\ x^2 + \beta 2\, x + \beta 3$

and so on..

Fit Linear models with NumPy

We will use two NumPy functions:

- **np.polyfit** is a function that performs the linear fit based on the **Least-Squares method** and returns the fitted parameters of a polynomial

- **np.poly1** it constructs a polynomial function from the parameters returned by polyfit. It constructs and returns the fitted model function.

```
coeff_array=np.polyfit(xarray, yarray, deg)
```

xarray, yarray - 1D arrays
deg: integer number specify the degree of the polynomial
coeff_array: 1D array containing the numerical values of fitted parameters: β0, β1, etc.

```
model_func = np.poly1d(coeff_array) #fitted model function
yp=model_func(xarray) #returns predicted yp values from the model
```

You can use directly both in one line of code:

```
model_func = np.poly1d(np.polyfit(xarray, yarray, deg))
```

Example – Fit a linear model with NumPy

step1 - Look at the raw data and choose the model

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt


#create data
x_data = np.arange(1,25)
y_data = x +6 * np.random.random(x.size)**2-1


#plot the raw data
fig, ax= plt.subplots()
ax.plot(x_data,y_data,'Dr',label='data')
ax.set_xlabel('x')
ax.set_ylabel('y')


#choose the model
#the model is y = β0* x + β1
```

Example – Fit a linear model with NumPy

step2 - Perform the fit and construct the fitting model function

```
#the model is y = β0* x + β1
coeff=np.polyfit(x, y, 1)
mymodel = np.poly1d(coeff)


print(mymodel.c)  #access the coefficients (fitted parameters)
[0.95374719 0.97329153] #[β0,  β1]

print(mymodel.order) #the order of the polynomial, its degree
1


#Create 1D array of predicted values yp for each element of x.
yp = mymodel(x)

#add the fitting model to the same Axes as the row data plot
ax.plot(x,yp,'b',label='model')
```

Example – Fit a linear model with NumPy

step3 - Calculating the R^2

$$R^2 = 1 - \frac{\sum_i (y_i - y_{ip})^2}{\sum_i (y_i - \bar{y})^2} = 1 - \frac{SSR}{SST}$$

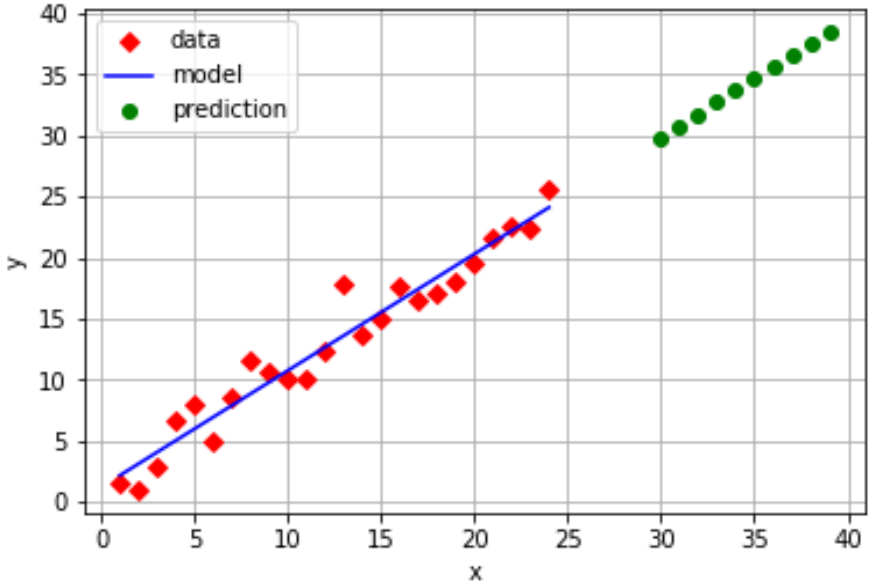
```
#You can calculate the R2 by applying the formula
SSR = ((y_data - yp)**2).sum()
SST = ((y_data - y_data.mean())**2).sum()

R2 = 1-(SSR/SST)
print(R2)
0.966040933792211


#or you can import the r2_score function from scikit-learn, the
Python machine Learning library

from sklearn.metrics import r2_score
print(r2_score(y,yp))
0.966040933792211
```

Example – Fit a linear model with NumPy

step4 - Make Predictions

If the model is good, you can use it to make predictions.

```
# predict y values for x values 30-40
xpred=np.arange(30,40)
ypred=mymodel(xpred)
ax.plot(xpred,ypred,'go',label='prediction')
ax.legend()
ax.grid()
plt.show()
```



To install **scikit-learn module :**

if you use Anaconda - type at the IPython Console or bash terminal

```
conda install scikit-learn
```

If you use standard Python3 IDLE:

```
pip3 install scikit-learn
```

To import it use:

```
import sklearn
```

The scikit-learn is a Python machine leaning library https://scikit-learn.org/stable/
You can also perform linear regression with scikit-learn or with SciPy.

Non-linear models

Non-Linear models are functions that are non-linear in their parameters

Exponential , power, Gaussian, Fourier etc.. functions, are non-linear because their coefficients are not linear

Exponential $y = yo \exp(-k x)$

Power $y = a\, x^b$

Examples of non-linear models and their applications:

https://acsess.onlinelibrary.wiley.com/doi/10.2134/agronj2012.0506

look at table1 for a summary of non-linear models

Non-linear least-squares procedure

Fit non-linear models with SciPy

For non-linear models, we will use the scipy.optimize module, and in particular the curve_fit() function , which implements a **non-linear least-squares procedure**.

Import the curve_fit() in this way:

```
from scipy.optimize import curve_fit
```

```
popt, pcov = curve_fit(func, xarray, yarray, guess_array)
```

func - the fitting model function, which can be a custom function defined by def
xarray – array-like (1D array, Series) the independent variable
yarray – array-like (1D array, Series) the dependent variable
initial_guess – a list or 1D array containing the initial guess for each paramater
Useful when curve_fit() cannot calculate the covariance matrix.

The function returns:
popt array– fitting coefficients – 1D array
Optimal values for the parameters so that the sum of the square residuals is minimized.

pcov matrix – covariance matrix – 2D array NxN where N is the number of fitting parameters.

Calculate the Residuals for each point.

$r_i = y_i - y_{ip}$

```
yp= func(x_data, *popt) #predicted values
residuals = y_data - yp #the * operator means all the elements
in variable popt.
```

Calculate the standard error of the parameters

the square root of the diagonal elements of the covariance matrix gives the standard error for each parameter

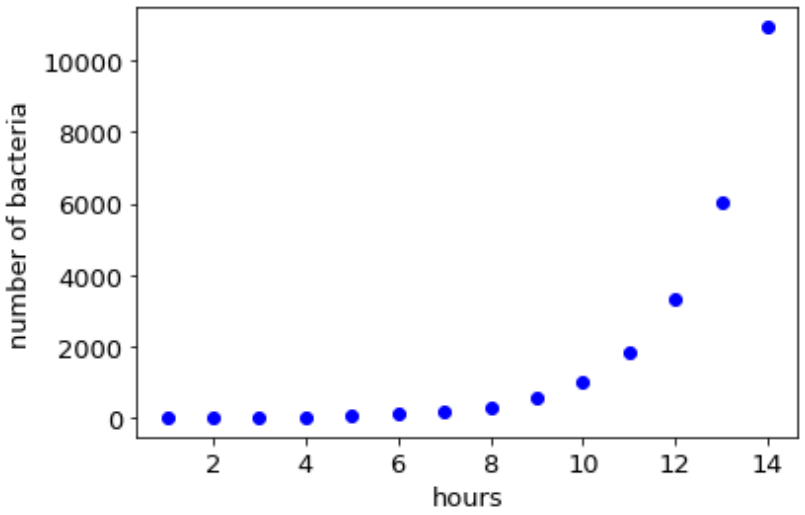
```
perr = np.sqrt(np.diag(pcov))
```

Example – Fit a non-linear model with SciPy

Download the data file bacteria.csv from Canvas DATA, which contains data on bacterial growth. The 1st field is the hours since observation and the 2nd field the number of bacteria in the sample. We will fit the data with an exponential growth model.

$$y = a\, e^{bx}$$

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

data=np.loadtxt('bacteria.csv',delimiter=',')
x=data[:,0]  #hours since observation
y=data[:,1]  #number of bacteria


#plot row data
plt.rcParams['font.size']=13
fig, ax=plt.subplots()
ax.plot(x, y,'ob', label='data'
ax.set_xlabel('hours')
ax.set_ylabel('number of bacteria')
```

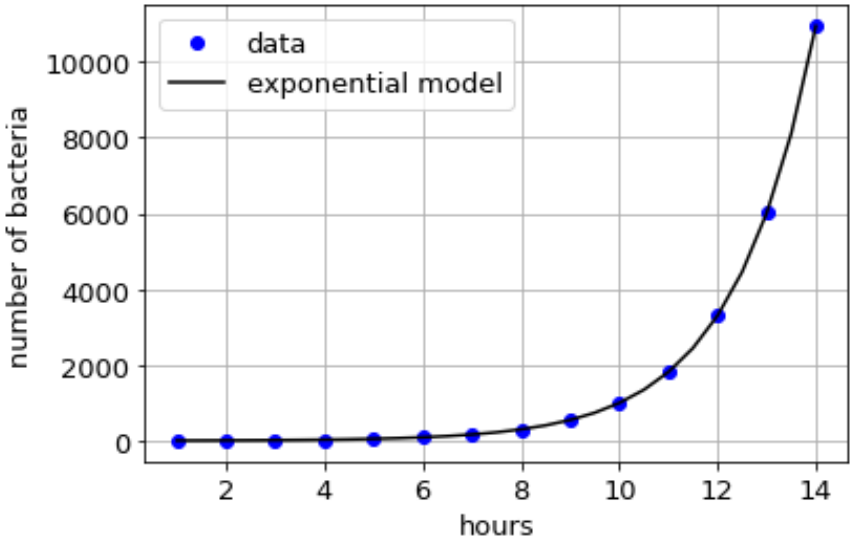


Example – Fit a non-linear model with SciPy


```
#choose the model, and make the function model
def mymodel(x,a,b): #the independent variable (x) must be listed first
    return a*(np.exp(b*x))


#fit with curve_fit
popt, pcov = curve_fit(mymodel, x, y)

#Sometimes an initial guess of the parameters is required for the
fitting method to work.
#popt, pcov = curve_fit(mymodel, x, y, [2.1, 0.3])) #provide an
initial guess


print(popt)
[2.49652581 0.59892497] #fitted parameters a and b

The order of the parameters in popt and pcov matches the order of the
parameters in the mymodel function


print(pcov) #covariance matrix
[[ 7.07224351e-05 -2.08093101e-06]
 [-2.08093101e-06  6.14351150e-08]]
```

Example – Fit a non-linear model with SciPy

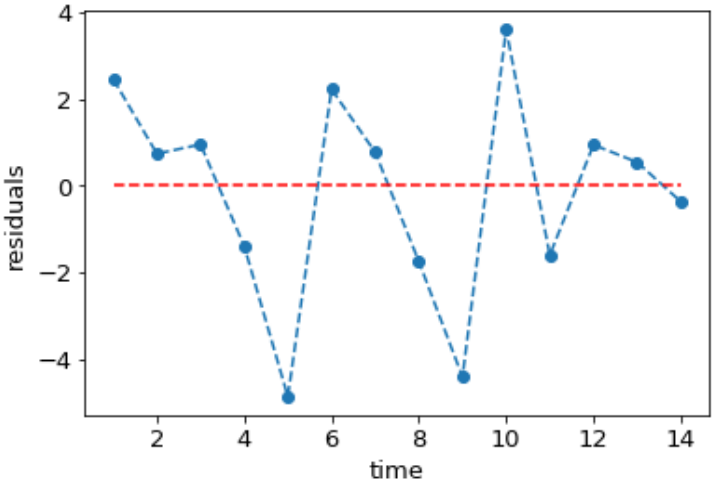
```
#Use the function to create an array containing the predicted values.

xsmall=np.arange(np.min(x),np.max(x)+0.1,0.5) #make more x values to smooth the model function

ypsmall=mymodel(xsmall,*popt) #the * operator means all the elements in variable popt.

#plot the fitted model.
ax.plot(xsmall,ypsmall,'k-',label='exponential model')
ax.grid()
ax.legend()
```



Non-linear models - Evaluating the fit

You can calculate the Residuals and make diagnostic plots.

```
#calculate and plot the residuals
yp=mymodel(x,*popt)
residuals = y - yp

fig1, ax1=plt.subplots()
ax1.plot(x, residuals,'o--')

#make and plot a line y=0
ax1.plot(x,np.zeros(x.size),'--r')
ax1.set_xlabel('time')
ax1.set_ylabel('residuals')
```



"good" fit: values are distributed randomly around 0, and there is no cluster of values all positive or all negative.

Non-linear models - Calculating standard errors of the parameters

The covariance matrix is:

```
print(pcov)
[[ 7.07224351e-05 -2.08093101e-06]
 [-2.08093101e-06  6.14351150e-08]]
```

To calculate the standard error for each parameter, we calculate the square root of the diagonal elements of the covariance matrix.

```
perr = np.sqrt(np.diag(pcov))
print(perr)
[0.00840966 0.00024786]
```

```
print("a=%.4f std_err=%.4f" %(popt[0],perr[0]))
print("b=%.4f std_err=%.4f" %(popt[1],perr[1]))
a=2.4965 std_err=0.0084
b=0.5989 std_err=0.0002
```

The estimated parameters are:

a=2.4965±0.0084

b=0.5989±0.0002

Make predictions

We can calculate how many bacteria are expected after 15 hours.

```
xpred=15
ypred=mymodel(xpred,*popt)

ax.plot(xpred,ypred,'dg', label='prediction')
ax.legend()

plt.show()
```