

# Key Modules

- The **Python interactive shell** has a number of built-in functions. They are loaded automatically as a shell starts and are always available, such as `print()` and `input()` for I/O, number conversion functions `int()`, `float()`, `complex()`, data type conversions `list()`, `tuple()`, `set()`, etc.
- In addition to the above built-in functions, a large number of pre-defined functions are also available as a part of libraries bundled with Python distributions. These functions are defined in **modules** are called built-in modules.
- Built-in modules are written in C and integrated with the Python shell.

*math module*

# math module

- Mathematical calculations are an essential part of most Python development. Whether you're working on a scientific project, a financial application, or any other type of programming endeavour, you just can't escape the need for math.
- For straightforward mathematical calculations in Python, you can use the built-in mathematical operators, such as addition (+), subtraction (-), division (/), and multiplication (\*). But more advanced operations, such as exponential, logarithmic, trigonometric, or power functions, are not built in. Does that mean you need to implement all of these functions from scratch?
- Fortunately, no. Python provides a module specifically designed for higher-level mathematical operations: the math module.

## Importing the math module

- Since the math module comes packaged with the Python release, you don't have to install it separately. Using it is just a matter of importing the module:

```
■ >>> import math
```

- You can import the Python math module using the above command. After importing, you can use it straightaway.

# Constants of the math Module

- The Python math module offers a variety of predefined constants. Having access to these constants provides several advantages.
- For one, you don't have to manually **hardcode** them into your application, which saves you a lot of time. Plus, they provide consistency throughout your code.

# Constants of the math Module

## ● The module includes several famous mathematical constants and important values:

- **Pi** - pi has an infinite number of decimal places, but it can be approximated as  $22/7$ , or 3.141.
- **Tau** - Tau ( $\tau$ ) is the ratio of a circle's circumference to its radius. This constant is equal to  $2\pi$ , or roughly 6.28.
- **Euler's number** - Euler's number ( $e$ ) is a constant that is the base of the **natural logarithm**, a mathematical function that is commonly used to calculate rates of growth or decay. As with pi and tau, Euler's number is an irrational number with infinite decimal places. The value of  $e$  is often approximated as 2.718.
- **Infinity** - Infinity can't be defined by a number. Rather, it's a mathematical concept representing something that is never-ending or boundless. Infinity can go in either direction, positive or negative.
- **Not a number (NaN)** - Not a number, or NaN, isn't really a mathematical concept. It comes from the computer science field as a reference to values that are not numeric. A NaN value can be due to invalid inputs, or it can refer to a value that has not been computed yet.

**math.nan constant** is a predefined constant, which is defined in the **math** module, it returns floating-point **nan** (Not a Number), which is equivalent to `float('nan')`. **math.nan constant** is available from **Python 3.5**

# Mathematical Functions in Python

In python several mathematical operations can be performed with ease by importing a module named “math” which defines various functions which makes our tasks easier.

- **1. `ceil()`** :- This function returns the smallest integral value greater than the number. If number is already integer, same number is returned.
- **2. `floor()`** :- This function returns the greatest integral value smaller than the number. If number is already integer, same number is returned.



```
import math
```

```
a=2.3
```

```
# returning the ceil of 2.3
```

```
print("The ceil of 2.3 is :", end="")
```

```
print(math.ceil(a))
```

```
# returning the floor of 2.3
```

```
print("The floor of 2.3 is :", end="")
```

```
print(math.floor(2.3))
```

Output:

```
The ceil of 2.3 is : 3
```

```
The floor of 2.3 is : 2
```

```
import math
```

```
print(math.ceil(2.3))
```

```
print(math.floor(2.3))
```

```
print(math.ceil(2))
```

```
print(math.ceil(2.5))
```

```
print(math.ceil(2.8))
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
PS C:\Users\Angela\Desktop\PythonScripts\A  
on.exe c:/Users/Angela/Desktop/PythonScrip
```

```
3
```

```
2
```

```
2
```

```
2
```

```
3
```

```
2
```

# Mathematical Functions in Python

- ③ **3. fabs()** :- This function returns the absolute value of the number.
- ④ **4. factorial()** :- This function returns the factorial of the number.  
An error message is displayed if number is not integral.

```
import math
a= -10
b= 5

print ("The fabs of -10 is : ", end="")
print (math.fabs(a))

print ("The factorial of 5 is : ", end="")
print (math.factorial(b))
```

Output:

```
The absolute value of -10 is : 10.0
The factorial of 5 is : 120
```

# Mathematical Functions in Python

- **5. `copysign(a, b)` :-** This function returns the number with the value of 'a' but with the sign of 'b'. The returned value is float type.
- **6. `gcd(a, b)` :-** This function is used to compute the greatest common divisor of 2 numbers passed in its arguments. This function works in python 3.5 and above.

```
import math
a= -10;b= 5; c=15;5

#returning the copysigned value.
print ("The copysigned value of -10 and 5.5 is : ", end="")
print (math.copysign(5.5,-10))

#returning the gcd of 15 and 5
print ("The gcd of 5 and 15 is : ", end="")
print (math.gcd(5,15))
```

Output:

```
The copysigned value of -10 and 5.5 is : -5.5
The gcd of 5 and 15 is : 5
```

# Mathematical Functions in Python

- **7. `exp(a)` :-** This function returns the value of  $e$  raised to the power  $a$  ( $e**a$ ). (Note :  $e$  is the Euler's constant = 2.718...)
- **8. `log(a, b)` :-** This function returns the logarithmic value of  $a$  with base  $b$ . If base is not mentioned, the computed value is of natural log.

```
import math

# returning the exp of 4
print ("The e**4 value is : ", end="")
print (math.exp(4))

# returning the log of 2,3
print ("The value of log 2 with base 3 is : ", end="")
print (math.log(2,3))
```

Output:

```
The e**4 value is : 54.598150033144236
The value of log 2 with base 3 is : 0.6309297535714574
```

# Mathematical Functions in Python

- **9. `pow(a, b)` :-** This function is used to compute value of *a* raised to the power *b* (`a**b`).
- **10. `sqrt()` :-** This function returns the square root of the number.

```
#importing "math" for mathematical operations
import math

# returning the value of 3**2
print ("The value of 3 to the power 2 is : ", end="")
print (math.pow(3,2))

# returnig the square root of 25
print ("The value of square root of 25 : ", end="")
print (math.sqrt(25))
```

Output:

```
The value of 3 to the power 2 is : 9.0
The value of square root of 25 : 5.0
```

# Mathematical Functions in Python

- **11. `math.fsum`** – used to find the sum of all the items of an iterable such as list, tuple, and set. It always returns a float value.

```
import math

a = math.fsum ([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
b = math.fsum ((10, 20, 30, 40, 50))
c = math.fsum ({1.1, 2.2, 3.3, 4.4, 5.5})

print (a)
print (b)
print (c)
```

# Python sum

- The `sum()` is a built-in Python function that takes an iterable as an argument, adds the elements of an iterable, and returns the sum.
- The iterators can be anything like List, Tuple, Set, or Dictionary.



The syntax of the `sum()` function is:

```
sum(iterable, start)
```

The `sum()` function adds `start` and items of the given `iterable` from left to right.

---

## **sum() Parameters**

- **iterable** - iterable (list, tuple, dict, etc). The items of the iterable should be numbers.
- **start** (optional) - this value is added to the sum of items of the iterable. The default value of `start` is 0 (if omitted)

```
marks =[65, 71, 68, 74, 61]
```

```
# find sum of all marks  
total_marks = sum(marks)  
print(total_marks)
```

**# Output: 339**

```
numbers = [2.5, 3, 4, -5]
```

```
#start parameter is not provided  
numbers_sum = sum(numbers)  
print(numbers_sum)
```

```
# start = 10  
numbers_sum = sum(numbers, 10)  
print(numbers_sum)
```

## Output

4.5  
14.5

# Mathematical Functions in Python

- ④ **12. degrees()** :- This function is used to convert argument value from radians to degrees.
- ④ **13. radians()** :- This function is used to convert argument value from degrees to radians.

```
import math
a = math.pi/6
b = 30

# returning the converted value from radians to degrees
print("The converted value from radians to degrees is :", end="")
print (math.degrees(a))

# returning the converted value from degrees to radians
print("The converted value from degrees to radians is :", end="")
print (math.radians(a))
```

Output:

```
The converted value from radians to degrees is : 29.999999999999996
The converted value from degrees to radians is : 0.5235987755982988
```

*random module*

# random module

- Even for someone not interested in computer programming, the usefulness of generating random numbers in certain circumstances is something obvious. In most board games we throw dice to generate an unpredictable number that defines the player's next move. Also, we can all agree that playing any card game would be pointless without a proper shuffle between rounds.
- But random numbers are not only important in relatively trivial fields like entertainment or gambling. They're especially crucial in the field of cryptography. In order to ensure safe transmission of data, every time a secure connection is necessary, a random key has to be generated. Many different kinds of electronic communication use this kind of security. It's very important for the key to be difficult to guess - the best way to ensure that is by making it random since the moment someone guesses the key, they are able to decipher the message - and the communication is not secure anymore.

# Python Random Module

- Python, obviously offers a super easy-to-use toolkit to handle random numbers.
- A module, called random, implements a pseudo-random number generator, and contains methods that let us directly solve many different programming issues where randomness comes into play.

# The random() Method

- The most important method of the random module is the random() method.
- The random() method generates a **random float in range (0,1), exclusive of 0 and 1**

```
import random
```

```
random.random()
```

```
0.8474337369372327
```

# uniform()

- If you need to generate random **floats that lie within a specific [x, y] interval**, you can use **random.uniform().**,

**Exclusive (x and y)**

```
import random

random.uniform(20, 30)
random.uniform(30, 40)
```

```
>>> random.uniform(20, 30)
27.42639687016509
>>> random.uniform(30, 40)
36.33865802745107
```



# Generate Random Integers - randint()

- The random.randint() method returns a random integer **between the specified integers(inclusive of the integers)**

Example: randint()

```
import random

random.randint(1, 100)
random.randint(1, 100)
```

# Generate Random Numbers within Range

- The `random.randrange()` method returns a randomly selected element from the range created by the `start`, `stop` and `step` arguments.

## Syntax

```
random.randrange(start, stop, step)
```

## Parameter Values

Parameter	Description
<i>start</i>	Optional. An integer specifying at which position to start. Default 0
<i>stop</i>	Required. An integer specifying at which position to end.
<i>step</i>	Optional. An integer specifying the incrementation. Default 1

```
import random

print("Genrating random number within a given range")
# Random number between 0 and 29
number1 = random.randrange(30)
print("Random interger: ", number1)

# Random number between 10 and 29
number2 = random.randrange(10, 30)
print("Random integer: ", number2)

# Random number between 25 and 200 divisible by 5
number3 = random.randrange (25, 201, 5)
print("Random integer: ", number3)
```

# Select Random Elements - random.choice()

- The random.choice() method returns a randomly selected element from a non-empty sequence.
- An empty sequence as argument raises an IndexError.

```
import random  
random.choice('computer')  
  
random.choice([12,23,45,67,65,43])  
  
random.choice((12,23,45,67,65,43))
```

Example:

```
>>> import random  
>>> random.choice('computer')  
't'  
>>> random.choice([12,23,45,67,65,43])  
45  
>>> random.choice((12,23,45,67,65,43))  
67
```

# Shuffle Elements Randomly

- The `random.shuffle()` method randomly reorders the elements in a list.
- **Note:** This method changes the original list, it does not return a new list.

```
import random
numbers=[12,23,45,67,65,43]

random.shuffle(numbers)
print(numbers)

random.shuffle(numbers)
print(numbers)
```

## Example:

```
>>> numbers=[12,23,45,67,65,43]
>>> random.shuffle(numbers)
>>> numbers
[23, 12, 43, 65, 67, 45]
>>> random.shuffle(numbers)
>>> numbers
[23, 43, 65, 45, 12, 67]
```

# sample()

```
random.sample(seq, n)
```

- The sample() function takes two arguments, and *both are required*.
- It picks up a random subsequence from the given sequence and returns it as the output.
- The first parameter is a sequence and the second is an integer value specifying how many values need to be returned in the output.
- It raises a TypeError if you miss any of the required arguments.

# Examples

```
print(random.sample([1,2,3,4,5,6,7,8,9],4))
```

**OUTPUT:** [1, 4, 5, 9]

```
import random

aSet = ("Jhon", "Kelly", "Scoot", "Emma", "Eric")
sample_set = random.sample(aSet, 3)
print(sample_set)
```

```
import random

examplet_list = [20, 40, 20, 20, 60, 70]
sampled_list2= random.sample(examplet_list, 4)
print(sampled_list2)
```

## Summary – functions in random module

Function	Description
randrange()	Can return random values between the specified limit and interval
randint()	Returns a random integer between the given limit
choice()	Returns a random number from a sequence
shuffle()	Shuffles a given sequence
sample()	Returns randomly selected items from a sequence
uniform()	Returns floating-point values between the given range
random()	Returns floating-point values between 0 and 1



## *datetime module*

# The datetime module

- Dealing with dates and times can be a hassle. Thankfully in Python, there's a built-in way of making it easier: the Python datetime module.
- The datetime module supplies classes for manipulating dates and times.
- These classes provide several functions to deal with dates, times and time intervals.
- Date and datetime are an object in Python, so when you manipulate them, you are manipulating objects and not string or timestamps.

# The datetime module

Before jumping into writing code, it's worth looking at the four main object classes that are used in the datetime module. Depending on what we're trying to do, we'll likely need to make use of one or more of these distinct classes:

- **date** – Allows us to manipulate dates independent of time (month, day, year).
- **time** – Allows us to manipulate time independent of date (hour, minute, second, microsecond).
- **datetime** – Allows us to manipulate times and dates together (month, day, year, hour, second, microsecond).
- **timedelta**— A duration of time used for manipulating dates and measuring.

# date class

- Used to instantiate date objects in Python.
- When an object of this class is instantiated, it represents a date in the format **YYYY-MM-DD**.
- Constructor of this class needs three mandatory arguments year, month and day.

## Constructor syntax:

```
class datetime.date(year, month, day)
```

# Creating a date Object

**All arguments are required.** Arguments must be integers, in the following ranges:

- ☐  $\text{MINYEAR} \leq \text{year} \leq \text{MAXYEAR}$
- ☐  $1 \leq \text{month} \leq 12$
- ☐  $1 \leq \text{day} \leq \text{number of days in the given month and year}$
- ☒ If an argument outside those ranges is given, `ValueError` is raised.
- ☒ A date object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar.
  - ☐ January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. 2

# Creating a date object

```
import datetime

d= datetime.date(2019,4,13)
print(d)
```

When you run the program, the output will be:

```
2019-04-13
```

```
from datetime import date

a=date(2019,4,13)
print(a)
```

`date()` in this example is a constructor of the `date` class. The constructor takes three arguments: year, month and day.

The variable `d/a` is a date object.

# import datetime to get the today's Date

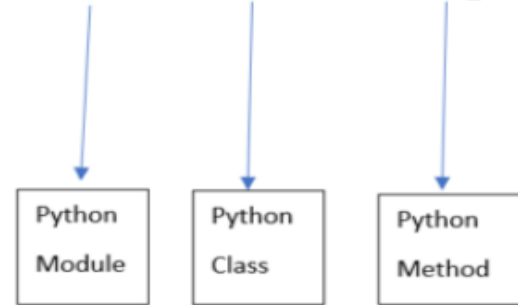
- The `date.today()` function fetches the current date from the system and represents the same.
- Here, we imported the `date` class from the `datetime` module. Then, we used the `date.today()` method to get the current local date. By the way, `date.today()` returns a `date` object, which is assigned to the `today_date` variable in the program.

```
import datetime
today_date = datetime.date.today()
print(today_date)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Angela\Desktop\PythonScripts\Batch
c:/Users/Angela/Desktop/PythonScripts/Batch
2022-03-23
```

`datetime.date.today()`



using date.today()

- date.today function has several properties associated with it.
- We can print individual day/month/year, as shown on the right:

```
import datetime
todays_date= datetime.date.today()

print(todays_date)
print(todays_date.day)
print(todays_date.month)
print(todays_date.year)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Angela\Desktop\PythonScripts\Batch
c:/Users/Angela/Desktop/PythonScripts/Batch
2022-03-23
23
3
2022
```



# Returning the weekday

## `date.weekday()`

- Return the day of the week as an integer, where **Monday is 0** and Sunday is 6.

## `date.isoweekday()`

- Return the day of the week as an integer, where **Monday is 1** and Sunday is 7. For example, `date(2002, 12, 4).isoweekday() == 3`, a Wednesday.

- Monday-0
- Tuesday-1
- Wednesday-2
- Thursday-3
- Friday-4
- Saturday-5
- Sunday-6

# Example

```
import datetime
todays_date= datetime.date.today()

print(todays_date)
print(todays_date.day)
print(todays_date.month)
print(todays_date.year)
print("Week Day:", todays_date.weekday())
print("ISO Week Day:", todays_date.isoweekday())
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

```
PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine> &
c:/Users/Angela/Desktop/PythonScripts/Batch-5/Mine/test
2022-03-23
23
3
2022
Week Day: 2
ISO Week Day: 3
```

# date.replace() function

- At times, a situation may arise, when we would want to alter part of the date expression. This task can be achieved using replace() function.

## **Note:**

- The method will show an error if any of the parameters are out of range, or the new date is invalid. For example, February has 28( or 29) days so if you enter 29 for a non-leap year, it will show a ValueError.

# date.replace() function

## Syntax:

```
replace(year=self.year, month=self.month, day=self.day)
```

## Parameter(s):

- **year**: new year value of the instance (range:  $1 \leq \text{year} \leq 9999$ )
- **month**: new month value of the instance (range:  $1 \leq \text{month} \leq 12$ )
- **day**: new day of the instance (range:  $1 \leq \text{day} \leq 31$ )

If values are not in the given range a *ValueError* is raised.

## Return value:

The return type of this method is a date class object after replacing the parameters.

# replace()

## ● Syntax:

```
date.replace(year,day,month)
```

## Example:

```
from datetime import datetime,date

dt = date.today()
print("Current date: ", dt)

res= dt.replace(year=2021,day=20)
print("Modified date:", res)
```

## Output:

```
Current date:  2020-07-26
Modified date:  2021-07-20
```

# The time class

- Python has a module named **time** to handle time-related tasks. To use functions defined in the module, we need to import the module first.

```
import time
```

Creation of time objects:

- Calling the constructor of the time class can create the time objects.

Time Constructor:

```
time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None,  
*, fold=[0,1])
```

# The time class

- All the parameters are optional parameters.
- The tzinfo is an object of a class derived from tzinfo which represents the timezone
- The fold parameter specifies whether there was any fold in time. A fold in time means a reverse back of the clock time. In countries following **Daylight Saving** time during the end of summer clocks are reversed back by 1 hour. This reverse back is a fold in time.
- \* operator - a tuple can be unpacked, and a time object can be constructed out of the values from the tuple.
- The time objects can be created, by calling the constructor of time class without providing any parameters. A time object without any parameters will have its hour, minute, second and microsecond initialized to zero.

# time class

- A time object instantiated from the time class represents the local time.

```
from datetime import time

#time(hour =0, minute=0, second=0)
a= time()
print("a=",a)

#time(hour, minute and second)
b=time(11, 34, 56)
print("b=",b)

#time(hour, minute and second)
c=time(hour=11, minute=34, second=56)
print("c=",c)

#time(hour, minute, second, microsecond)
d= time(11, 34,56,234566)
print("d= ",d)
```

When you run the program, the output will be:

```
a = 00:00:00
b = 11:34:56
c = 11:34:56
d = 11:34:56.234566
```



# Create a time object using python's splat operator:

```
import datetime

# All the time parameters for creating a time instance as a tuple
# 11PM, 59 Minutes, 59 Seconds, 999999 microseconds
timeTuple = (11,59,59,9999)

# Use the splat operator to unpick the time elements and create a time instance
timeInstance = datetime.time(*timeTuple)

# print the time instance constructed by using splat operator
print(timeInstance)
```

## Output:

```
11:59:59.999999
```

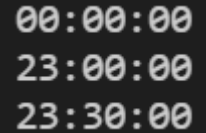
# Create a time object without any parameters and modify it, using replace()

```
import datetime

#Crare a time with all the components as 0
t1= datetime.time()
print(t1)

# Crate a new time object with hour as 23 that is 11 PM
t2= t1.replace(23)
print(t2)

#Create a new tome object with hour as 23 and minute as 30 that is 11.30PM
t3= t2.replace(minute=30)
print(t3)
```



00:00:00  
23:00:00  
23:30:00

# Print hour, minute, second and microsecond

- Once you create a time object, you can easily print its attributes such as hour, minute etc.

```
from datetime import time

a = time(11, 34, 56)

print("hour", a.hour)
print("minute", a.minute)
print("second", a.second)
print("microsecond", a.microsecond)
```

```
from datetime import time

a = time(11, 34, 56)

print("hour =", a.hour)
print("minute =", a.minute)
print("second =", a.second)
print("microsecond =", a.microsecond)
```

When you run the example, the output will be:

```
hour = 11
minute = 34
second = 56
microsecond = 0
```

# The datetime class

- DateTime class of the DateTime module as the name suggests contains information on both date as well as time.

## Constructor:

- `class datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, fold=0)`
- The **year, month and day arguments are required**. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments must be integers in the ranges, as shown on the right.
- If an argument outside those ranges is given, ValueError is raised.
  - `MINYEAR <= year <= MAXYEAR,`
  - `1 <= month <= 12,`
  - `1 <= day <= number of days in the given month and year,`
  - `0 <= hour < 24,`
  - `0 <= minute < 60,`
  - `0 <= second < 60,`
  - `0 <= microsecond < 1000000,`
  - `fold in [0, 1].`

## Creating Datetime Objects

- Since datetime is both a module and a class within that module, we'll start by **importing the datetime class from the datetime module**.
- Then, we'll print the current date and time to take a closer look at what's contained in a datetime object.
- We can do this using datetime's **.now() function**. We'll print our datetime object, and then also print its type, so we can take a closer look.

# Creating an instance of DateTime class

```
from datetime import datetime

#Initializing constructor
a= datetime(2022,10,22)
print(a)

# Initializing constructor with time parameters as well
a = datetime (2022,10,22,6,2,32,5456)
print(a)
```

```
2022-10-22 00:00:00
2022-10-22 06:02:32.005456
```

# Datetime class

```
import datetime

now_date = datetime.datetime.now()
print(now_date)
print(type(now_date))
```

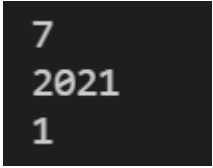
```
2021-07-01 15:04:48.321423
<class 'datetime.datetime'>
```

# Getting elements from the date object

- Anyway, we were hoping to separate out specific elements of the date for our analysis. One way can do that using the built-in class attributes of a datetime object, like `.month` or `.year` and `.day`:

```
import datetime
now_date = datetime.datetime.now()

print(now_date.month)
print(now_date.year)
print(now_date.day)
```



7  
2021  
1



## Constants

- The datetime module has the following constants:
- **datetime.MINYEAR**
  - The smallest year number allowed in a date or datetime object. MINYEAR is 1.
- **datetime.MAXYEAR**
  - The largest year number allowed in a date or datetime object. MAXYEAR is 9999.

# Instance methods

## **datetime.date()**

- Return date object with year, month and day.

## **datetime.time()**

- Return time object with hour, minute, second, microsecond.

```
import datetime

dt = datetime.datetime(2021, 6, 3, 12, 37, 45, 10000)
print(dt.date())
print(dt.time())
```

# datetime.combine()

- **datetime.combine()** combines instances of datetime.date and datetime.time into a single datetime.datetime instance.

```
from datetime import date, time, datetime

today = date.today()
print("Today :", today)

now = datetime.now()
print("Now :", now)

current_time = time(now.hour, now.minute, now.second)
print("Current Time:", current_time)

combined_datetime_obj = datetime.combine(today, current_time)
print("Combined datetime :", combined_datetime_obj)
```

```
Today : 2021-07-05
Now : 2021-07-05 14:40:56.117761
Current Time : 14:40:56
Combined datetime : 2021-07-05 14:40:56
```

# datetime.timedelta

- Python datetime module contains the timedelta(), which is used to perform date and time manipulations in python.
- A timedelta object **represents the difference between two dates or times**. It **represents a duration**, i.e., a duration of date or time, or a span of time.

# timedelta object

A timedelta object is returned when we call the timedelta constructor as shown below:

```
td_object =timedelta(days=0, seconds=0, microseconds=0,  
milliseconds=0, minutes=0, hours=0, weeks=0)
```

- All arguments are optional and default to 0 and can be floats, integers, positives, or negatives. You can perform mathematical operations such as addition, subtraction, and multiplication with the timedelta class.

# How to Get Past and Future Dates With Timedelta?

- Since timedelta represents a duration, to get any future or past date, you add or subtract timedelta to or from the current date.
- Here's a simple equation to show this, where n represents the number of days in integers:

```
import datetime
n=1
current_date = datetime.datetime.today()

past_date = datetime.datetime.today() - datetime.timedelta(days=n)

future_date = datetime.datetime.today() + datetime.timedelta(days=n)
```

# Use timedelta to Add Days

- Add Days to a date or datetime Object to get a future date.

```
>>> now + timedelta(days=3)
```



```
2020/11/3  
+ 3 days  
= 2020/11/6
```

```
import datetime

now= datetime.datetime.now()
print("Today's Date", now)

from datetime import timedelta

print("After 3 day:", now + timedelta(days=3))
print("Before 3 day:", now + timedelta(days=-3))
```

PROBLEMS

2

OUTPUT

DEBUG CONSOLE

TERMINAL

```
PS C:\Users\Angela\Desktop\PythonScripts\Classwork> & C:/U
/Users/Angela/Desktop/PythonScripts/Classwork/Mine/deltaex
Today's Date: 2021-07-05 15:22:00.134004
After 3 days:  2021-07-08 15:22:00.134004
Before 3 days: 2021-07-02 15:22:00.134004
```



# Difference between two dates and times

```
from datetime import datetime, date

t1 = date(year= 2018, month =7, day=12)
t2 = date(year= 2017, month =12, day=23)
t3= t1- t2
print("t3 =", t3)

t4= datetime(year= 2018, month =7, day=12, hour=7, minute =9, second=33)
t5= datetime(year= 2019, month =6, day=10, hour=5, minute =55, second=13)
t6 = t4 - t5
print("t6 =", t6)

print("type of t3 ", type(t3))
print("type of t6 ", type(t6))
```

```
t3 = 201 days, 0:00:00
t6 = -333 days, 1:14:20
type of t3 = <class 'datetime.timedelta'>
type of t6 = <class 'datetime.timedelta'>
```

Notice, both `t3` and `t6` are of `<class 'datetime.timedelta'>` type.

# Difference between two timedelta objects

```
from datetime import timedelta

t1 = timedelta(weeks=2, days=5, hours=1, seconds=33)
t2 = timedelta(days= 4, hours =11, minutes=4, seconds =54)
t3= t1- t2

print("t3 =", t3)
```

```
t3 = 14 days, 13:55:39
```

Here, we have created two `timedelta` objects `t1` and `t2`, and their difference is printed on the screen.

# Printing negative timedelta object

```
from datetime import timedelta

t1 = timedelta(seconds=33)
t2 = timedelta(seconds =54)
t3= t1- t2

print("t3 =", t3)
print("t3 =", abs(t3))
```

When you run the program, the output will be:

```
t3 = -1 day, 23:59:39
t3 = 0:00:21
```

# Time duration in seconds

You can get the total number of seconds in a timedelta object using `total_seconds()` method.

```
from datetime import timedelta

t = timedelta(days=5, hours=1, seconds=33, microseconds=233423)
print("total second =", t.total_seconds())
```

When you run the program, the output will be:

```
total seconds = 435633.233423
```

# Convert a timedelta to days, hours, and minutes

- A `datetime.timedelta` object contains days, seconds, and microseconds, but when called **returns the days and time of the object**.
- Getting the days from a timedelta is direct in Python. Call `timedelta.days` to return the number of days.
- Call `timedelta.seconds` to return the number of seconds.
- Use the result of `timedelta.total_seconds()` and the division symbol `//` to divide the seconds by 3600 to calculate the number of hours.
- Call `str(delta)`, where `delta` is a `datetime.timedelta` object to convert it into a string.

# Python format datetime

- Python has `strftime()` and `strptime()` methods to format date objects.
- **strptime** means string **parser**, this will parse and convert a string object to datetime object.
- **strftime** means string **formatter**, this will format a datetime object to string object.

# strftime() - datetime object to string

- The strftime() method is defined under classes date, datetime and time.
- The method creates a formatted string from a given date, datetime or time object.
- The syntax of strftime() method is...

```
dateobject.strftime(format)
```

Where format is the desired format of date string that user wants. Format is built using codes shown in the table in the next slide...

# Format Codes

## **Code** **Meaning**

%a	Weekday as Sun, Mon
%A	Weekday as full name as Sunday, Monday
%w	Weekday as decimal no as 0,1,2...
%d	Day of month as 01,02
%b	Months as Jan, Feb
%B	Months as January, February
%m	Months as 01,02
%y	Year without century as 11,12,13
%Y	Year with century 2011,2012
%H	24 Hours clock from 00 to 23
%l	12 Hours clock from 01 to 12
%p	AM, PM
%M	Minutes from 00 to 59
%S	Seconds from 00 to 59
%f	Microseconds 6 decimal numbers



```
from datetime import datetime
now = datetime.now()
print(now)
```

```
print(now.strftime("%Y-%m-%d %H:%M:%S"))
print(now.strftime("%Y-%b-%d %H:%M:%S"))
print(now.strftime("%A-%d-%b, %Y %H:%M:%S"))
print(now.strftime("%m/%d/%Y, %H:%M:%S"))
print(now.strftime("%d/%b, %Y"))
print(now.strftime("%d/%B, %Y"))
print(now.strftime("%I%p"))
```

```
2022-03-28 16:27:45.206701
2022-03-28 16:27:45
2022-Mar-28 16:27:45
Monday, 28 Mar, 2022 16:27:45
03/28/2022, 16:27:45
28 Mar, 2022
28 March, 2022
04PM
```

# strptime()

- `strptime()` is used to parse string to datetime object.

`strptime(date_string, format)`

- example:

- `strptime("9/23/20", "%m/%d/%y")`

- Of course, `strptime()` isn't magic — it can't turn any string into a date and time, and it will need a little help from us to interpret what it's seeing!
- The format `"%m/%d/%y"` represents the corresponding "9/23/20" format. The output of the above command will be a Python datetime object.
- The format is constructed using pre-defined codes.

# String to datetime object

```
from datetime import datetime

date_string = "21 June, 2018"

print("date_string =", date_string)
print("type of date_string =", type(date_string))

date_object = datetime.strptime(date_string, "%d %B, %Y")

print("date_object =", date_object)
print("date_object =", type(date_object))
```

```
date_string = 21 June, 2018
type of date_string = <class 'str'>
date_object = 2018-06-21 00:00:00
type of date_object = <class 'datetime.datetime'>
```

In the above example:

● Here,

```
date_string = "21 June, 2018"  
... ..  
date_object = datetime.strptime(date_string, "%d %B, %Y")
```



- %d - Represents the day of the month. Example: 01, 02, ..., 31
- %B - Month's name in full. Example: January, February etc.
- %Y - Year in four digits. Example: 2018, 2019 etc.

# strptime()

- You may also have noticed that a time of 00:00:00 has been added to the date object.
- That's because we created a datetime object, which must include a date and a time. 00:00:00 is the default time that will be assigned if no time is designated in the string we're inputting.

# Format Code List

Directive	Meaning	Example
%a	Abbreviated weekday name.	Sun, Mon, ...
%A	Full weekday name.	Sunday, Monday, ...
%w	Weekday as a decimal number.	0, 1, ..., 6
%d	Day of the month as a zero-padded decimal.	01, 02, ..., 31
%-d	Day of the month as a decimal number.	1, 2, ..., 30
%b	Abbreviated month name.	Jan, Feb, ..., Dec
%B	Full month name.	January, February, ...

<code>%m</code>	Month as a zero-padded decimal number.	01, 02, ..., 12
<code>%-m</code>	Month as a decimal number.	1, 2, ..., 12
<code>%y</code>	Year without century as a zero-padded decimal number.	00, 01, ..., 99
<code>%-y</code>	Year without century as a decimal number.	0, 1, ..., 99
<code>%Y</code>	Year with century as a decimal number.	2013, 2019 etc.
<code>%H</code>	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23
<code>%-H</code>	Hour (24-hour clock) as a decimal number.	0, 1, ..., 23
<code>%I</code>	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
<code>%-I</code>	Hour (12-hour clock) as a decimal number.	1, 2, ... 12
<code>%p</code>	Locale's AM or PM.	AM, PM

# String to datetime object

```
from datetime import datetime

dt_string = "12/11/2018 09:15:32"

#Considering date is in dd/mm/yyyy format
dt_object1 = datetime.strptime(dt_string, "%d/%m/%Y %H:%M:%S")
print(" dt_object1 =", dt_object1)

#Considering date is in mm/dd/yyyy format
dt_object2 = datetime.strptime(dt_string, "%m/%d/%Y %H:%M:%S")
print(" dt_object2 =", dt_object2)
```

When you run the program, the output will be:

```
dt_object1 = 2018-11-12 09:15:32
dt_object2 = 2018-12-11 09:15:32
```



## ValueError in strptime()

- If the string (first argument) and the format code (second argument) passed to the `strptime()` doesn't match, you will get `ValueError`.
- For example:

```
from datetime import datetime

date_string = "12/11/2018"
date_object= datetime.strptime(date_string, "%d %m %Y")

print("date_object",date_object)
```

If you run this program, you will get an error.

```
ValueError: time data '12/11/2018' does not match format '%d %m %Y'
```

*sys*

# The sys Module

sys module - System-specific parameters and functions

- This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
- To work with the **sys module**, you must first import the **module**.

# sys.version

- sys.version – This stores the information about the current version of python.

```
>>> import sys
>>> sys.version
'3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)]'
>>> |
```

# sys.path

- sys.path – Path variable stores the directory path in the form of a list of strings. Whenever you import a module or run a program using a relative path, python interpreter searches for the necessary module or script using the path variable.

```
>>> import sys
>>> sys.path
['', 'C:\\Users\\Angela\\AppData\\Local\\Programs\\Python\\Python39\\Lib\\idlelib', 'C:\\Users\\Angela\\AppData\\Local\\Programs\\Python\\Python39\\python39.zip', 'C:\\Users\\Angela\\AppData\\Local\\Programs\\Python\\Python39\\DLLs', 'C:\\Users\\Angela\\AppData\\Local\\Programs\\Python\\Python39\\lib', 'C:\\Users\\Angela\\AppData\\Local\\Programs\\Python\\Python39', 'C:\\Users\\Angela\\AppData\\Roaming\\Python\\Python39\\site-packages', 'C:\\Users\\Angela\\AppData\\Local\\Programs\\Python\\Python39\\lib\\site-packages']
>>> |
```

# sys.argv

- **sys. argv** is a list in **Python**, which contains the command-line arguments passed to the script.
- It stores the script(file) name as the 1st value followed by the arguments we pass.
- argv values are stored as type string and you must explicitly convert it according to your needs.

# sys.executable

- sys.executable – Prints the absolute path of the python interpreter binary.

```
>>> sys.executable
'C:\\Users\\Angela\\AppData\\Local\\Programs\\Python\\Python39\\pythonw.exe'
>>> |
```



## sys.platform

- **sys.platform** – Prints the OS platform type. This function will be very useful when you run your program as a platform dependent.

```
>>> sys.platform  
'win32'  
>>> |
```

```
import sys  
sys.platform
```

System	platform value
AIX	'aix'
Linux	'linux'
Windows	'win32'
Windows/Cygwin	'cygwin'
macOS	'darwin'

# sys.exit

- **sys.exit** – The `sys.exit()` function allows the developer to exit from Python. The exit function takes an optional argument, typically an integer, that gives an exit status. Zero is considered a “successful termination”. We can either use an integer value as Exit Status or other kinds of objects like string(“Failed”) as shown in the below example.

```
import sys

if sys.platform == "darwin" : # if Macintosh
    #code goes her
    pass
else:
    print("This script is attended to only run on Macintosh, detective platform:",sys.platform)
    sys.exit("Failed")
```

# sys.copyright

- © This String just displays the copyright information on currently installed Python version. Let's execute this on the system by making a script:

```
>>> print(sys.copyright)
Copyright (c) 2001-2020 Python Software Foundation.
All Rights Reserved.
```

```
Copyright (c) 2000 BeOpen.com.
All Rights Reserved.
```

```
Copyright (c) 1995-2001 Corporation for National Research Initiatives
All Rights Reserved.
```

```
Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

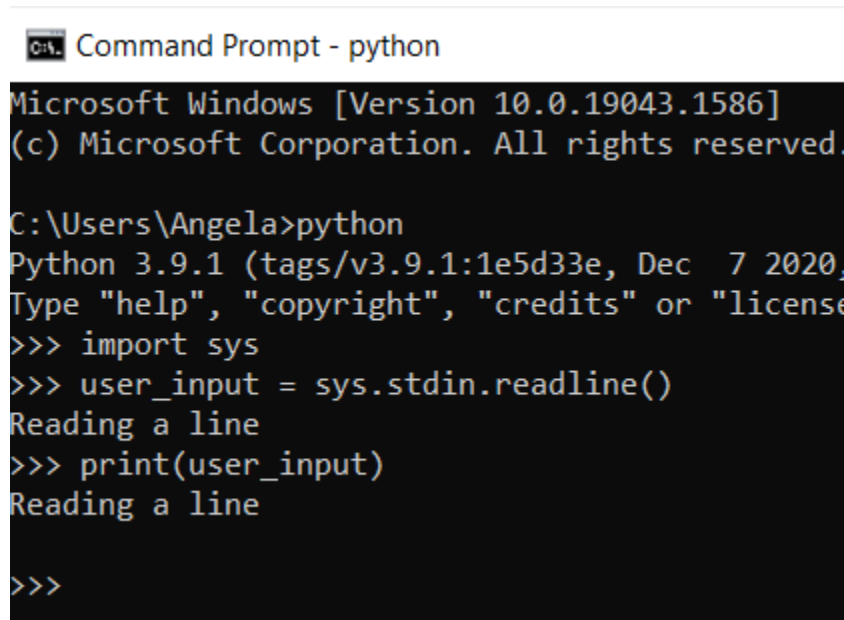
```
import sys
print(sys.copyright)
```

# Input and Output using sys

- Every serious user of a UNIX or Linux operating system knows standard streams, i.e., input, standard output and standard error. They are known as pipes. They are commonly abbreviated as **stdin**, **stdout**, **stderr**.
- The standard input (stdin) is normally connected to the keyboard, while the standard error and standard output go to the terminal (or window) in which you are working.
- These data streams can be accessed from Python via the objects of the sys module with the same names, i.e. **sys.stdin**, **sys.stdout** and **sys.stderr**.

# stdin

- stdin: It can be used to get input from the command line directly. It is used for standard input.
- It, also, reads the '\n' at the end of the input.



```
Command Prompt - python

Microsoft Windows [Version 10.0.19043.1586]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Angela>python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020)
Type "help", "copyright", "credits" or "license()"
>>> import sys
>>> user_input = sys.stdin.readline()
Reading a line
>>> print(user_input)
Reading a line
>>>
```

# stdout

- `stdout`: A built-in file object that is analogous to the interpreter's standard output stream in Python.
- `stdout` is used to display output directly to the screen console.
- `print(x)` is basically a shortcut for `sys.stdout.write(x + '\n')`

# Example

```
>>> import sys
>>> sample_input = ["Hi", "This is about stdout", "exit"]

for ip in sample_input:
    #Print to stdout
    sys.stdout.write(ip+ "\n")
```

Hi

3

This is about stdout

21

exit

5

# Redirecting stdout to a file

```
import sys

sys.stdout = open('out.txt', 'w')

print('The above command redirects the output to the file out.txt rather than the console')
```

To redirect the standard output to a file is to set `sys.stdout` to the file object, as shown above.

## ▼ MINE

- old\_sytle\_format.py
- out.txt
- print\_sep\_ex1.py
- rec\_demo.py
- recursion\_demo.py

≡ out.txt

```
1 The above command redirects the output to the file out.txt rather than the console
2
```



# stderr

- Python stderr is known as a standard error stream.
- It is similar to stdout because it also directly prints to the console but the main difference is that it is used to print error messages.

```
>>> import sys
>>> sys.stderr.write("This is error msg")
This is error msg17
>>> |
```

# Redirecting Error Information to a file

- Using `sys.stderr` and a text file in Python, we can log error information to the text file. See how:

```
import sys
```

```
23 sys.stderr = open('error.txt', 'w')  
24 raise Exception("This is an error")
```

```
error.txt  
1  Traceback (most recent call last):  
2  File "c:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine\sys_demo.py", line 24, in <module>  
3  |   raise Exception("This is an error")  
4  Exception: This is an error  
5
```

# sys.stdout - example

```
>>> sys.stdout.write("Hello")  
Hello5
```

- For every write() call the number of characters written, passed to the function respectively gets append to the output in console. Even passing an empty string results in an output of 0.
- This really only happens in a Python console, but not when executing a file with the same statements.
- Apart from writing out the given string, write will also return the number of characters (actually, bytes)

```
>>> ret = sys.stdout.write("Hello")  
Hello  
>>> print(ret)  
5
```