FUNCTION
CLASSES
MODULES

Seaborn Kusto

OFFICIAL TEAM

Malka
Rusyd
Abdussalam

Trianto
Haryo
Nugroho

Maulana
Ishaq
Siregar

Fauzi
Wardah
Ali

Rizki
Afrinal

FUNCTION (DEF)

- function is a group of related statements that performs a specific task
- a function is defined with the def keyword followed by the name of the function and its parameters in brackets()
- the code block in each function starts with a colon (:) and is indented
- The function stops when there is a return (expression) statement that returns (expression) to the caller

```
[1]  def square(x):  # x is a parameter (the input the function accepts)
       return x**2

[2]  square(3) # 3 is a argument (the thing we enter when calling the function)

     9
```

FUNCTION

```
[3]  def cube(x):
       cubic = x**3
       return cubic

[4]  cube(2)

     8

[5]  def squareroot(x):
       output = x**0.5
       return

[6]  squareroot(8)
```

- optionally, you can add a docstring (string documentation describing the context of the function)
- you can also make the function return no output with return none

FUNCTION

```
[7]  def div (x,y):
       division = x / y
       return division

[8]  div(10,5)

     2.0

[9]  div(y=10, x=5)

     0.5
```

- by default, python will position each parameter according to the order in which it was registered at the time it was defined and must be called in that order

- but you can also enter it in an unordered way by writing parameters and arguments

FUNCTION

```
[10] def minus(x,y):
       subtraction = x - y
       print('{} - {} = {}'.format(x,y,subtraction))
       return subtraction

[11] minus(2,3)

     2 - 3 = -1
     -1

[12] output = minus(2,3)
     print('the result of subtraction is {}'.format(output))

     2 - 3 = -1
     the result of subtraction is -1
```

- the return (expression) statement will make program execution exit the current function

FUNCTION

- all parameters (arguments) in python are 'passed by reference'. This means that when we change a variable, the data that refers to it will also change, both inside the function and outside the calling function. unless we do an assignment operation that will change the reference parameter

- something that is in a def function is local and can only be operated in a def function

```
def percent(percentage):
  percentage = 100*percentage
  return print(percentage,'%')
```

```
percentage
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-5dc9173ce437> in <module>()
----> 1 percentage

NameError: name 'percentage' is not defined

 SEARCH STACK OVERFLOW
```

```
percentage = 0.5
percent(percentage)
```

```
50.0 %
```

```
x = 0.2
percent(x)
```

```
20.0 %
```

CLASSES

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

CLASSES

- All classes have a function called *__init__()*, which is always executed when the class is being initiated.

- Use the *__init__()* function to assign values to object properties, or other operations that are necessary to do when the object is being created

```
class Person:

  def __init__(self, name, age):
    self.name = name
    self.age = age

p = Person("Kusto", 20)

print(p.name)
print(p.age)
```

```
Kusto
20
```

CLASSES

```
class Calculator:

  def __init__(self, num1, num2):
    self.num1 = num1
    self.num2 = num2

  def Multiple(self):
    self.mul = self.num1 * self.num2
    return self.mul

cal = Calculator("num1", "num2")
cal = Calculator(6, 5)
cal.Multiple()
```

```
30
```

Object Method

- Class object could be used to access different attributes.

- It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

CLASSES

```
class Calculator:

  def __init__(self, num1, num2):
    self.num1 = num1
    self.num2 = num2

  @classmethod
  def Multiple_1(cls, num1, num2):
    cls.mul_num = num1 * num2
    return cls.mul_num

cal = Calculator("num1", "num2")
cal.Multiple_1(5, 9)
```

```
45
```

Class Method

- A class method takes ***cls*** as the first parameter

- We use @classmethod decorator in python to create a class method

CLASSES

```
class Calculator:

  def __init__(self, num1, num2):
    self.num1 = num1
    self.num2 = num2

  @staticmethod
  def Multiple_2(num1, num2):
    mul = num1 * num2
    return mul

cal = Calculator.Multiple_2(6, 7)
print(cal)
```

```
42
```

Static Method

- Static method needs no specific parameters

- We use @staticmethod decorator to create a static method in python

PYDOC

The pydoc module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a web browser, or saved to HTML files. This can be done from the Anaconda Prompt. This code below is an example to generates documentation of the pass module.

```
Anaconda Prompt (Anaconda3)

(base) C:\Users\ASUS>python -m pydoc pass
The "pass" statement
**********************

   pass_stmt ::= "pass"

"pass" is a null operation ù when it is executed, nothing happens. It
is useful as a placeholder when a statement is required syntactically,
but no code needs to be executed, for example:

   def f(arg): pass    # a function that does nothing (yet)

   class C: pass       # a class with no methods (yet)
```

```
(base) C:\Users\ASUS>python /?
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I     : isolate Python from the user's environment (implies -E and -s)
-m mod : run library module as a script (terminates option list)
-O     : remove assert and __debug__-dependent statements; add .opt-1 before
         .pyc extension; also PYTHONOPTIMIZE=x
-OO    : do -O changes and also discard docstrings; add .opt-2 before
         .pyc extension
-q     : don't print version and copyright messages on interactive startup
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-u     : force the stdout and stderr streams to be unbuffered;
         this option has no effect on stdin; also PYTHONUNBUFFERED=x
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
         when given twice, print more information about the build
-W arg : warning control; arg is action:message:category:module:lineno
         also PYTHONWARNINGS=arg
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt : set implementation-specific option. The following options are available:
```

PYDOC

```
-X faulthandler: enable faulthandler
-X showrefcount: output the total reference count and number of used
    memory blocks when the program finishes or after each statement in the
    interactive interpreter. This only works on debug builds
-X tracemalloc: start tracing Python memory allocations using the
    tracemalloc module. By default, only the most recent frame is stored in a
    traceback of a trace. Use -X tracemalloc=NFRAME to start tracing with a
    traceback limit of NFRAME frames
-X showalloccount: output the total count of allocated objects for each
    type when the program finishes. This only works when Python was built with
    COUNT_ALLOCS defined
-X importtime: show how long each import takes. It shows module name,
    cumulative time (including nested imports) and self time (excluding
    nested imports). Note that its output may be broken in multi-threaded
    application. Typical usage is python3 -X importtime -c 'import asyncio'
-X dev: enable CPython's "development mode", introducing additional runtime
    checks which are too expensive to be enabled by default. Effect of the
    developer mode:
       * Add default warning filter, as -W default
       * Install debug hooks on memory allocators: see the PyMem_SetupDebugHooks() C function
       * Enable the faulthandler module to dump the Python traceback on a crash
       * Enable asyncio debug mode
       * Set the dev_mode attribute of sys.flags to True
       * io.IOBase destructor logs close() exceptions
-X utf8: enable UTF-8 mode for operating system interfaces, overriding the default
    locale-aware mode. -X utf8=0 explicitly disables UTF-8 mode (even when it would
    otherwise activate automatically)
-X pycache_prefix=PATH: enable writing .pyc files to a parallel tree rooted at the
    given directory instead of to the code tree
```

```
--check-hash-based-pycs always|default|never:
    control how Python invalidates hash-based .pyc files
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ';'-separated list of directories prefixed to the
               default module search path.  The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>;<exec_prefix>).
               The default module search path uses <prefix>\python{major}{minor}.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONUTF8: if set to 1, enable the UTF-8 mode.
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
PYTHONHASHSEED: if this variable is set to 'random', a random value is used
   to seed the hashes of str and bytes objects.  It can also be set to an
   integer in the range [0,4294967295] to get hash values with a
   predictable seed.
PYTHONMALLOC: set the Python memory allocators and/or install debug hooks
   on Python memory allocators. Use PYTHONMALLOC=debug to install debug
   hooks.
PYTHONCOERCECLOCALE: if this variable is set to 0, it disables the locale
   coercion behavior. Use PYTHONCOERCECLOCALE=warn to request display of
   locale coercion and locale compatibility warnings on stderr.
PYTHONBREAKPOINT: if this variable is set to 0, it disables the default
   debugger. It can be set to the callable of your debugger of choice.
PYTHONDEVMODE: enable the development mode.
PYTHONPYCACHEPREFIX: root directory for bytecode cache (pyc) files.
```

IO

The io module provides Python's main facilities for dealing with various types of I/O. There are three main types of I/O: text I/O, binary I/O and raw I/O. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a file object. Other common terms are stream and file-like object.

Text I/O expects and produces str objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.

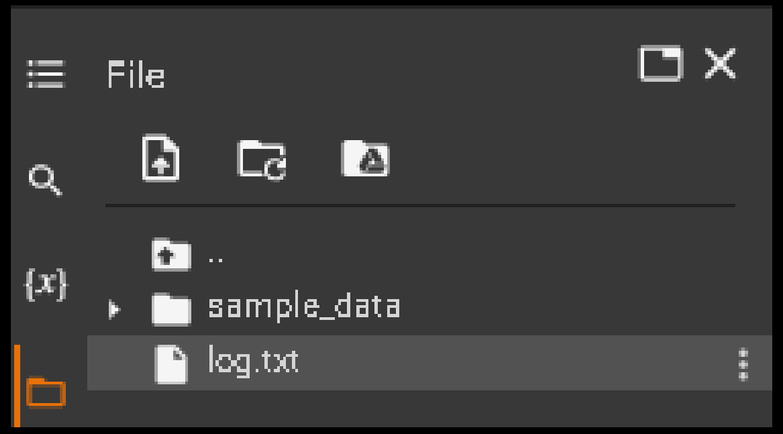
TXT FILE - WRITE

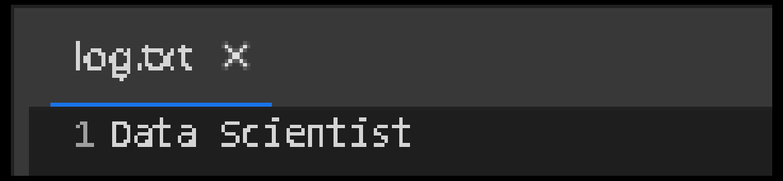
Write txt file

```
[ ]  # Import library io (input output) to be able to input/write in a new txt file

     import io
```

```
[ ]  # Wwrite txt in python without creating in notepad using with __ as __

     with open('log.txt', 'w') as writeFile:
       toLog = input("What do you want to write to the log? ")
       writeFile.write(toLog)

     # w for write
```

```
What do you want to write to the log? Data Scientist
```





IO

Read txt file

```
[ ]  # Read txt file about hobby using another format

     log = open('log.txt', 'r')
     logContents = log.read()
     print("The contents of log.txt is {}".format(logContents))

     # r for read
```

```
     The contents of log.txt is Data Scientist
```

TXT FILE - OPEN, READ & CLOSE

We can open, read and close txt file in python using open(), upload(), read() and close().

- open() used to open a txt file
- upload() used to upload a txt file
- read() used to read txt file.
- close() used to close a txt file.

TXT FILE - OPEN

Open

Open txt file from existing folder location using upload()

```
[ ]  # Open txt file from existing folder location using upload()

     from google.colab import files
     files.upload()
```

```
     Choose Files  No file chosen    Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
     Saving jackets.txt to jackets (1).txt
     {'jackets.txt': b'Product, Price\r\nBig Down Jacket, 100\r\nMedium Down Jacket, 80\r\nRegular Jacket, 40'}
```

```
[ ]  # Print the contents of the txt file

     print(workFileContents)
```

```
     Product, Price
     Big Down Jacket, 100
     Medium Down Jacket, 80
     Regular Jacket, 40
```

TXT FILE - READ

Read

Read txt file using read()

```
[ ]  # Read txt file using open(txt.file, 'r')

     workFile = open('jackets.txt', 'r')

     # 'r' for reading
```

```
[ ]  # Read txt file using read()

     workFileContents = workFile.read()
```

```
[ ]  # Read the first line of the txt file

     workFileFirstline = workFile.readline()
```

```
[ ]  # Print the first line of the txt file

     print(workFileFirstline)

     Product, Price
```

TXT FILE - CLOSE

Close

Close the txt file (delete temporary memory) using close()

```
[ ]  workFile.close()
```

TXT FILE - DELETE

To delete a file, you must import the OS module, and run its os.remove() function.

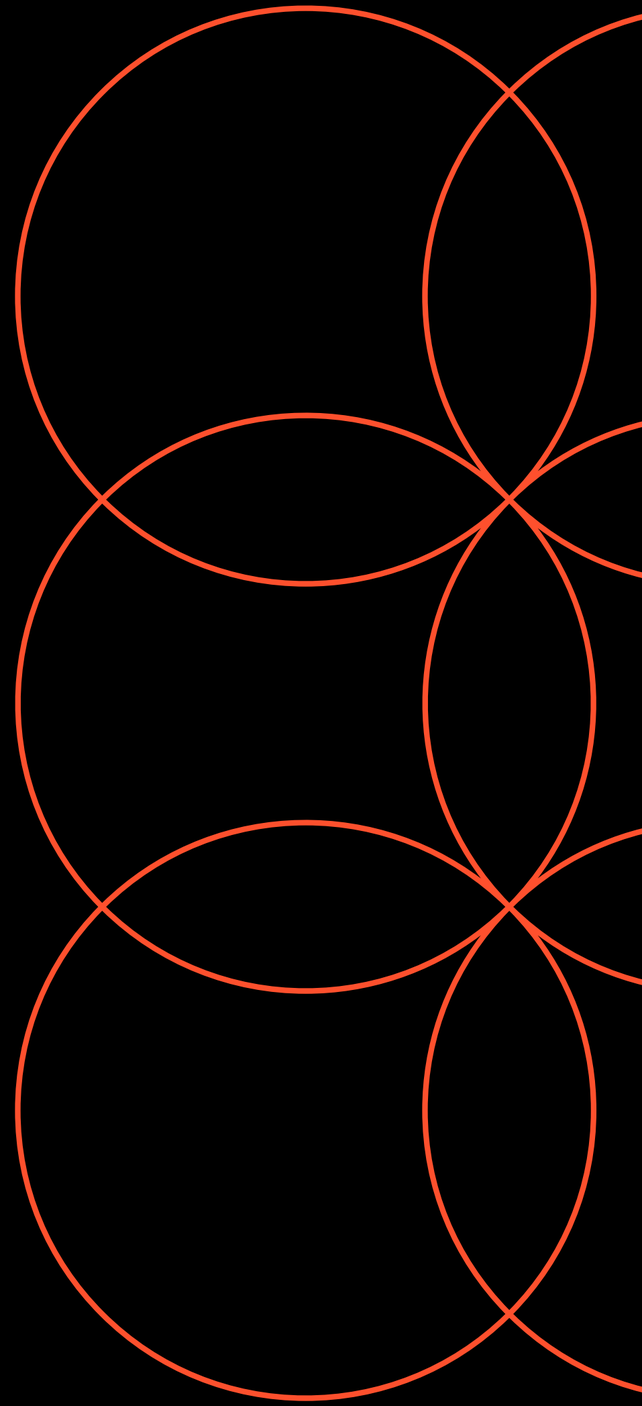
Remove the file "log_old.txt":

```
[ ] import os
```

Check if file exists, then delete it:

```
[ ] if os.path.isfile('log_old.txt'):
      os.remove('log_old.txt')
      print("the log_old file has been remove")
    else:
      print('there was no log_old file to remove')
```

```
there was no log_old file to remove
```

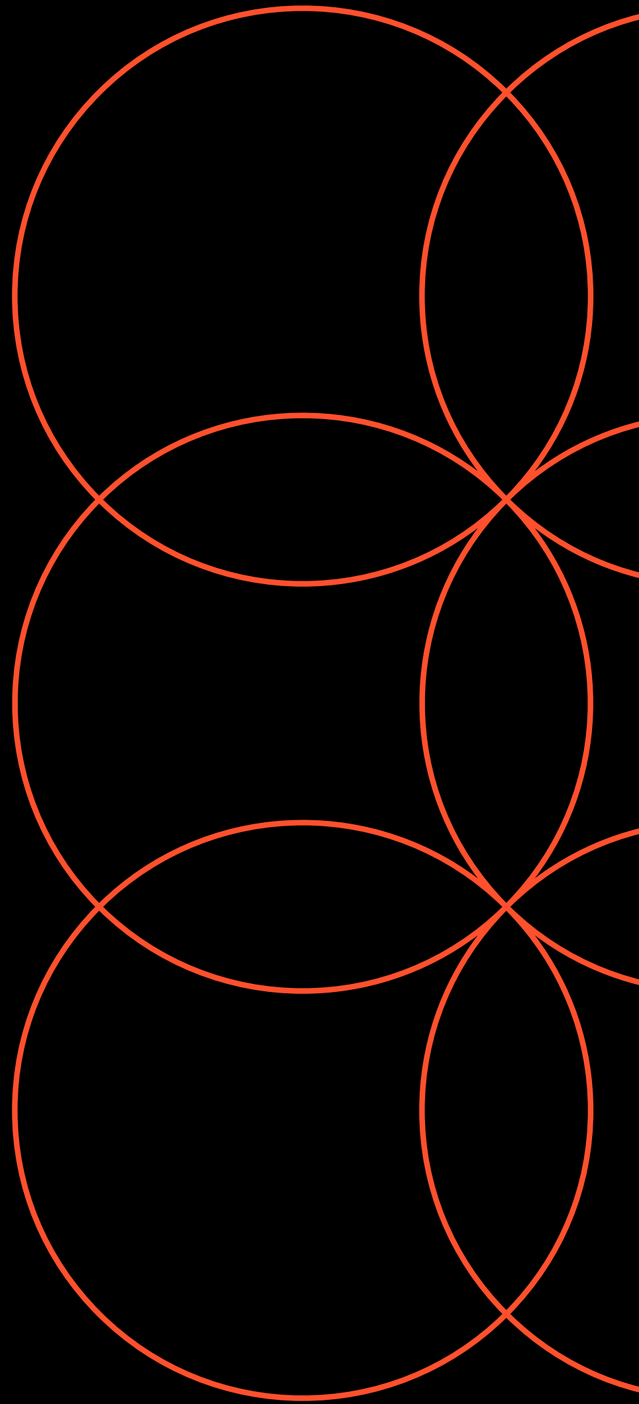


UNIT TEST

Tests written using the unittest module can help you find bugs in your programs, and prevent regressions from occurring as you change your code over time. Teams adhering to test-driven development may find unittest useful to ensure all authored code has a corresponding set of tests

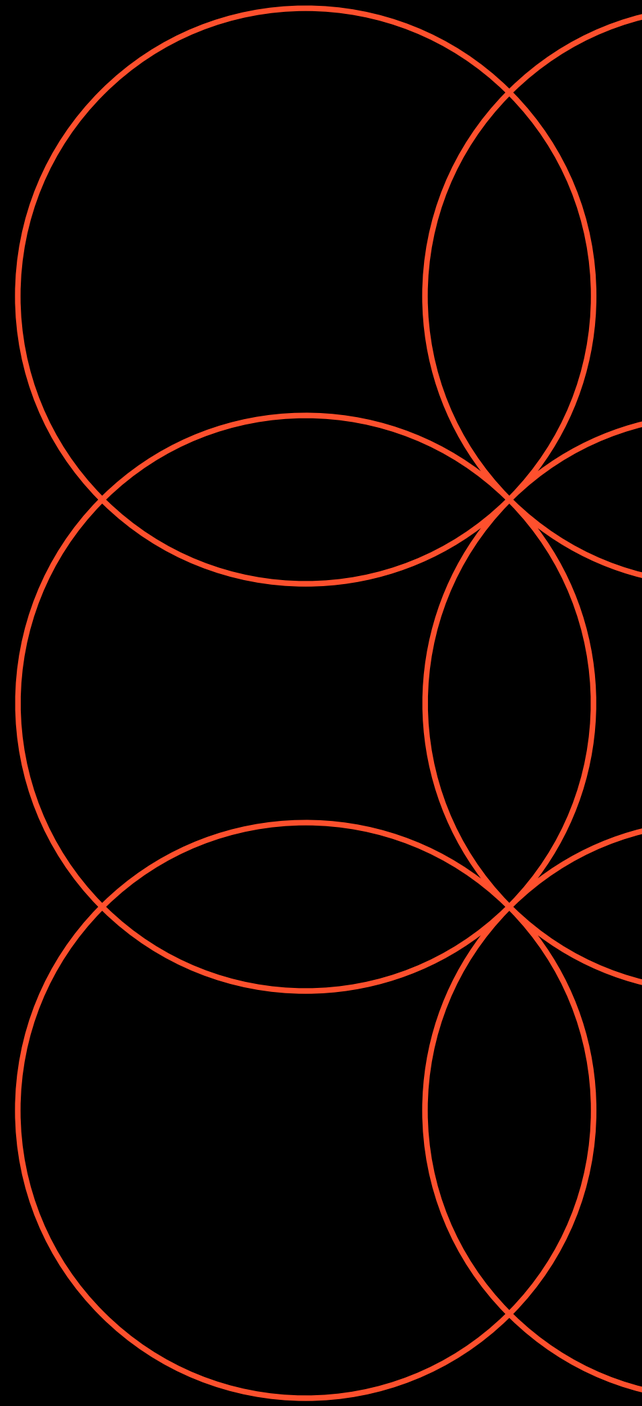
test case

- A test case is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.



UNIT TEST

```
[ ]  import unittest

     class SimpleTest(unittest.TestCase):

       # Returns True or False.
       def test(self):
         self.assertTrue(True)

     if __name__ == '__main__':
       unittest.main()
```

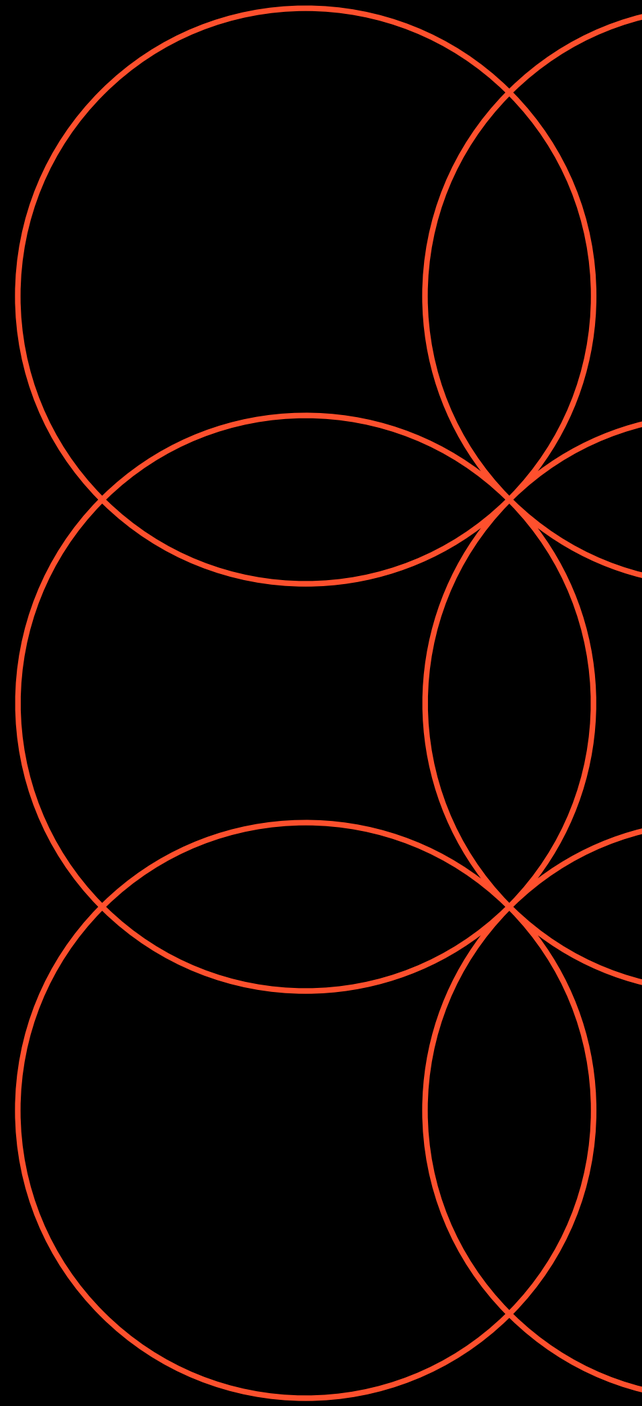


OS.PATH.ISFILE()

This module provides a portable way of using operating system dependent functionality. os.path module is sub module of OS module in Python used for common path name manipulation.

```
[ ] import os

    if os.path.isfile('log.txt'):
      writeFile = open('log.txt', 'a')
    else:
      witeFile = open('log.txt', 'w')

    toLog = input("What do you want to write to the log?")
    writeFile.write('\n'+toLog)
    writeFile.close()
```

```
What do you want to write to the log?hai
```



OS - CREATE DIRECTORY

The OS module in Python provides functions for interacting with the operating system. OS comes under Python's standard utility modules. This module provides a portable way of using operating system-dependent functionality.

```
[ ] import os
```

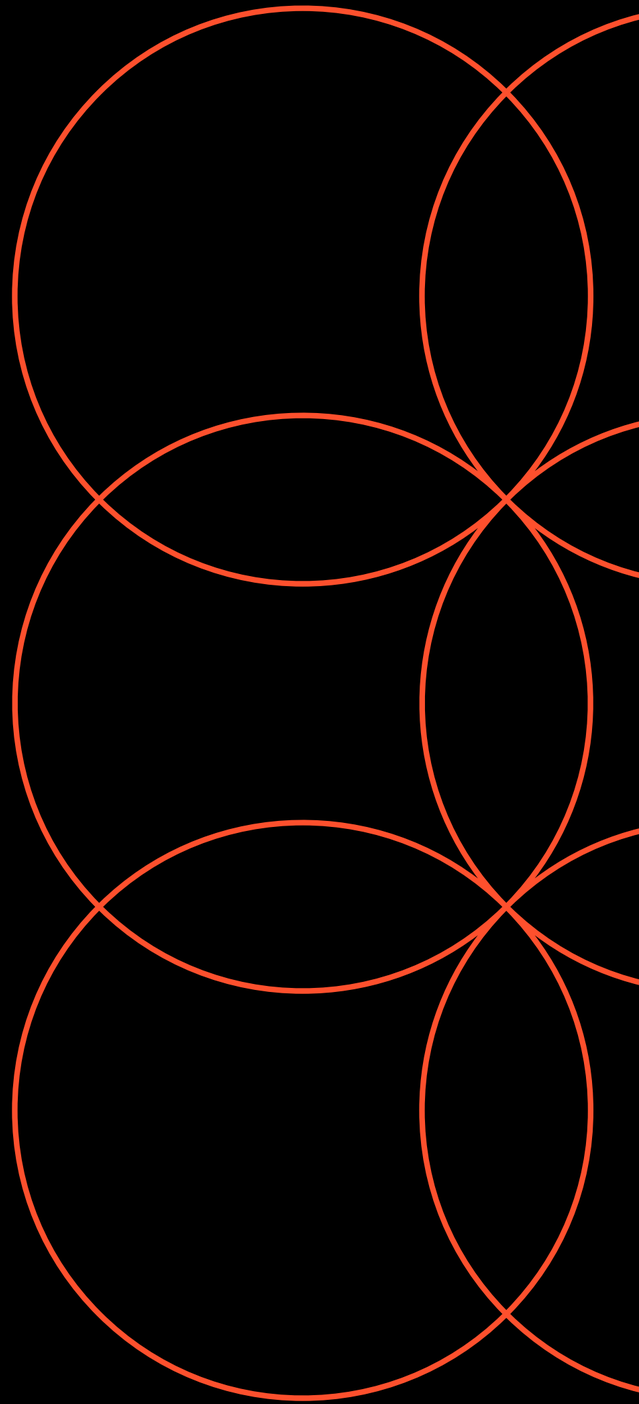
```
[ ] dirName = input("Enter the name of the folder you want to create")
    os.mkdir(dirName)
    print("Directory created")
```

```
Enter the name of the folder you want to createData Science
Directory created
```

MATH

This module provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the cmath module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.



MATH.PI

math.pi

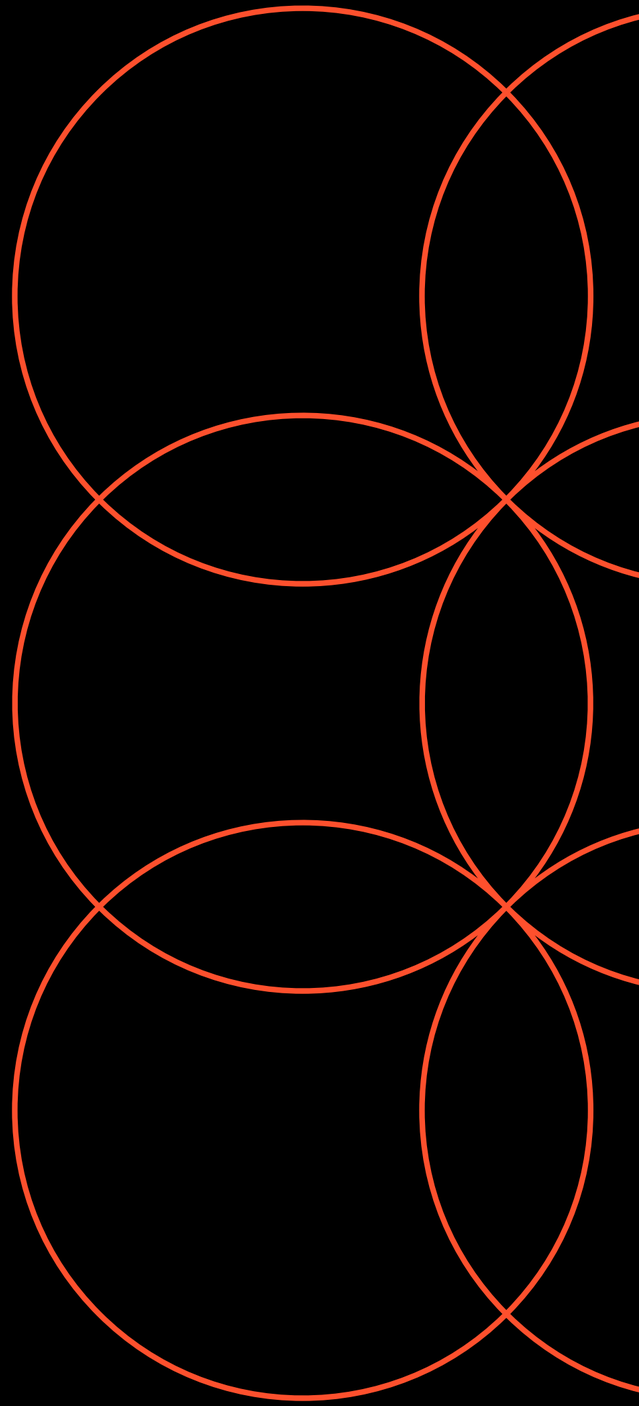
Calling The mathematical constant π = 3.141592..., to available precision.

```
[ ]  # Import math library

     import math
```

```
[ ]  # Assign math.pi as variable pi

     pi = math.pi
```

```
[ ]  # Calling the value of the pi variable

     pi
```

```
     3.141592653589793
```



MATH.CEIL(X)

math.ceil(x)

Return the ceiling of x, the smallest integer greater than or equal to x. If x is not a float, delegates to x.**ceil**, which should return an Integral value.

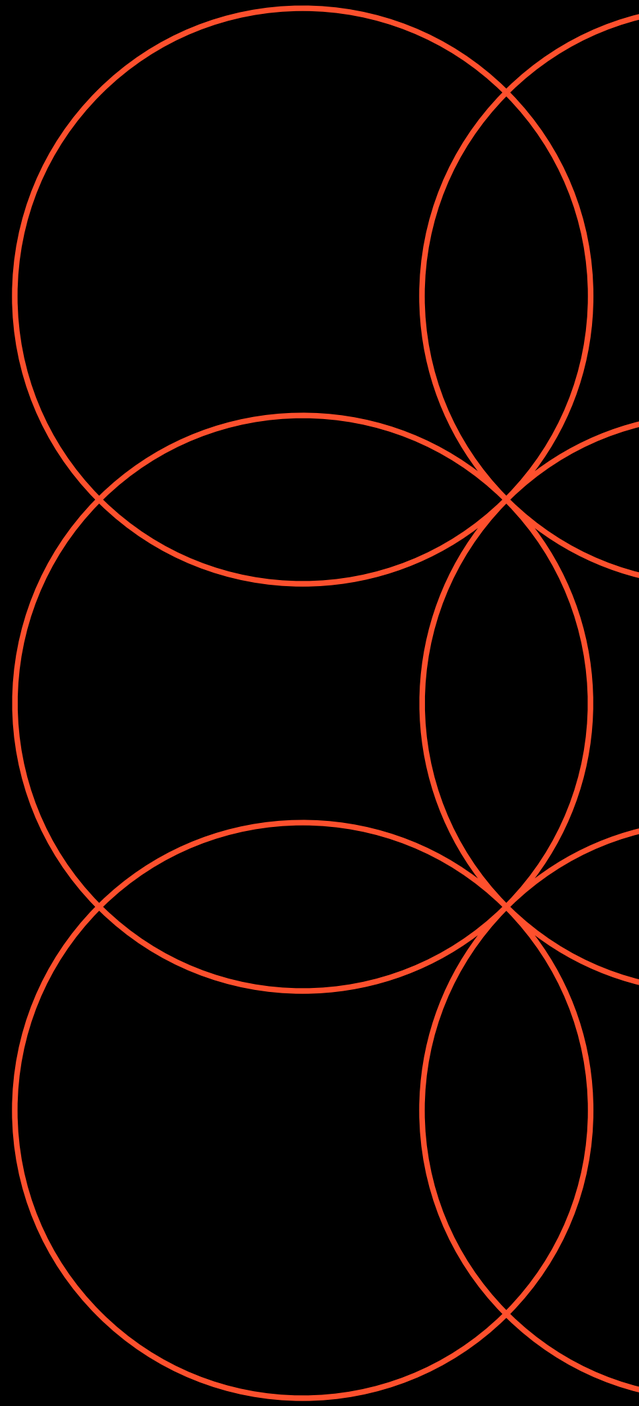
```
[ ]  # Assign math.ceil(pi) (to upper bound the value of pi) to a variable upperBound

     upperBound = math.ceil(pi)
```

```
[ ]  # Calling the value of the upperBound variable

     upperBound
```

```
4
```

```
[ ]  # Upper bound the float 9.99 using math.ceil(9.99)

     math.ceil(9.99)
```

```
10
```



MATH.FLOOR(X)

math.floor(x)

Return the floor of x, the largest integer less than or equal to x. If x is not a float, delegates to x.**floor**, which should return an Integral value.

```
[ ]  # Assign math.floor(pi) (to lower bound the value of pi) to a variable lowerBound

     lowerBound = math.floor(pi)
```

```
[ ]  # Calling the value of the lowerBound variable

     lowerBound

     3
```

```
[ ]  # lower bound the float 9.99 using math.floor(9.99)

     math.floor(9.99)

     9
```

DATETIME

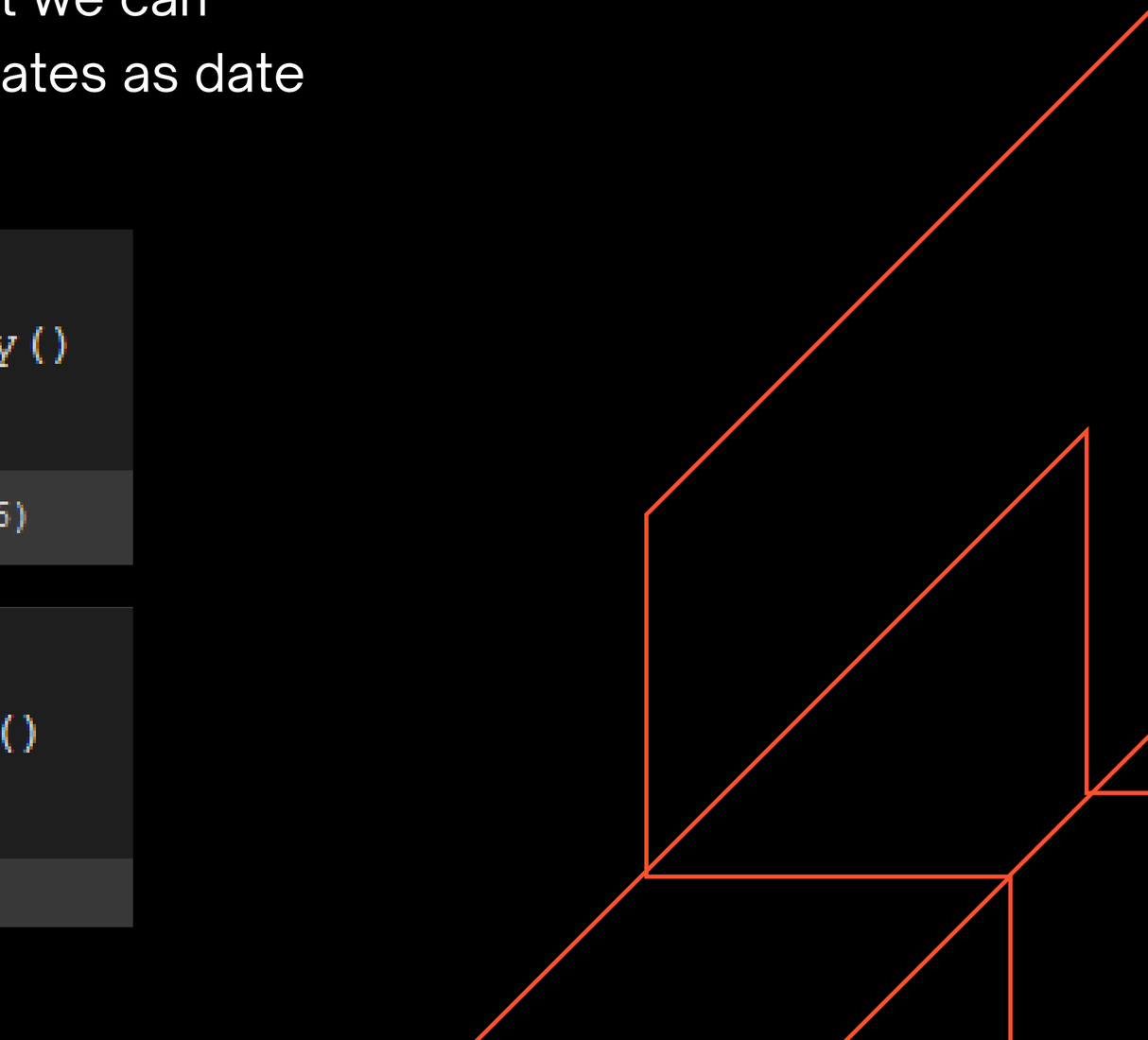
A date in Python is not a data type of its own, but we can import a module named **datetime** to work with dates as date objects

```
import datetime
todayWithTime = datetime.datetime.today()
todayWithTime
```

```
datetime.datetime(2022, 4, 8, 8, 56, 27, 243846)
```

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

```
datetime.date(2022, 4, 8)
```



DATETIME

- To create a date, we can use the **datetime()** class (constructor) of the datetime module.

- The **datetime()** class requires three parameters to create a date: year, month, day.

```
import datetime
birthday = datetime.date(2000, 3, 14)
print(birthday)
```

```
2000-03-14
```

DATETIME

- The datetime object has a method for formatting date objects into readable strings.
- The method is called **strftime()**, and takes one parameter, format, to specify the format of the returned string

```
import datetime
formatting_date = datetime.datetime.strftime(todaywithtime, "%m/%d/%Y")
formatting_date
```

```
'04/08/2022'
```

```
import datetime
formatting_time = datetime.datetime.strftime(todaywithtime, "%H : %M : %S")
formatting_time
```

```
'08 : 55 : 47'
```

RANDOM - RANDINT

- randint() is an inbuilt function of the random module in Python. The random module gives access to various useful functions and one of them being able to generate random numbers, which is randint().
- randint can use negative number parameter

```
[1]  from random import randint
```

```
[2]  for i in range(10):
       print(f'random num {i} is {randint(1,10)}')
```

```
random num 0 is 2
random num 1 is 6
random num 2 is 4
random num 3 is 2
random num 4 is 2
random num 5 is 2
random num 6 is 2
random num 7 is 1
random num 8 is 10
random num 9 is 3
```

```
[3]  number = []
     for i in range(10):
       number.append(randint(-10,10))
     print(number)
```

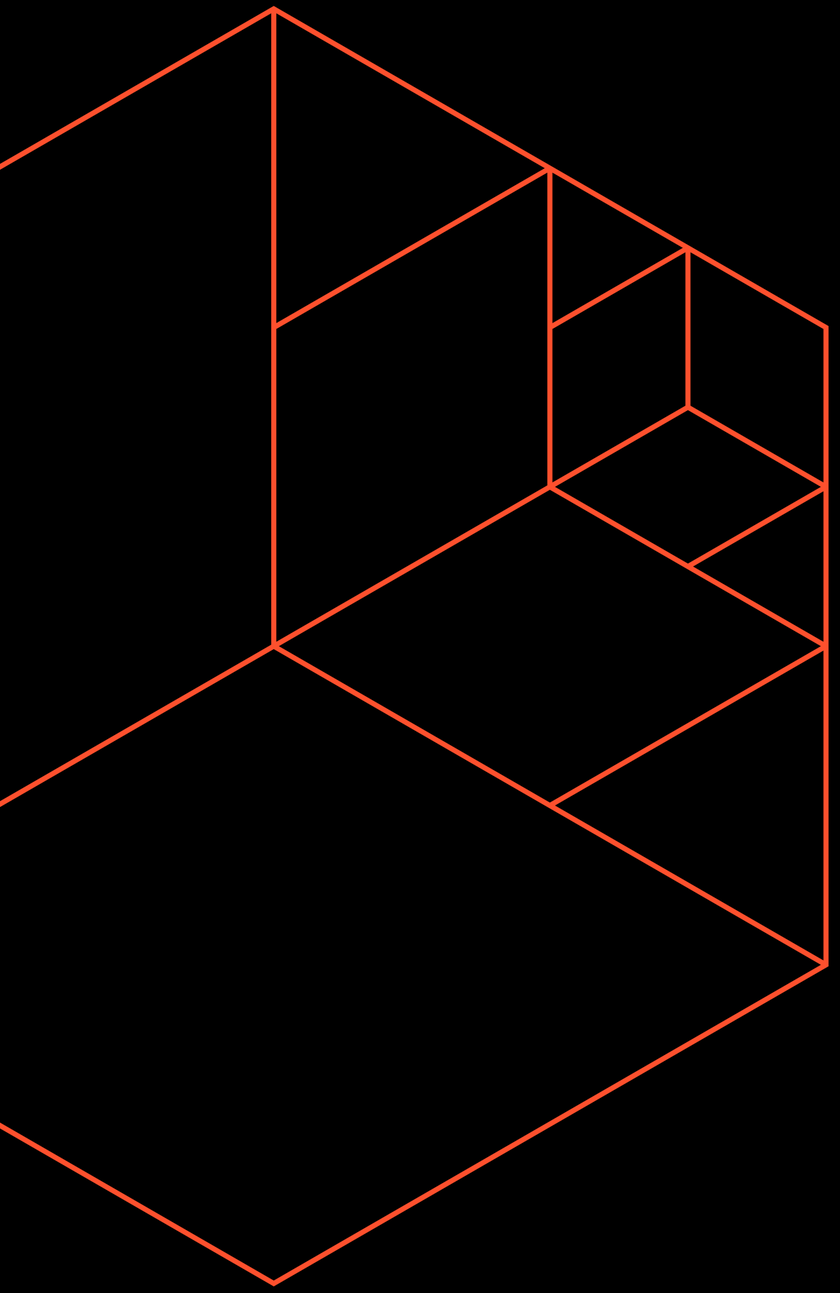
```
[3, -5, -9, -3, 4, 9, -6, -4, -4, -3]
```

RANDOM - RANDINT

- arguments in parameter in randint must be integer type value

```
number = []
for i in range(10):
  number.append(randint(-10,10.1)) #parameter must be integer type value
print(number)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-75ea69aa0433> in <module>()
      1 number = []
      2 for i in range(10):
----> 3   number.append(randint(-10,10.1)) #parameter must be integer type value
      4 print(number)
```



THANK YOU!