PYTHON PROGRAMMING

(18IS5DEPYP)

Unit - 4

Mrs. Bhavani K

Assistant Professor

Dept. of ISE, DSCE

Unit - 4

- **DEBUGGING, TESTING AND PROFILING:**
  - Debugging:
    - Dealing with Syntax Errors
    - Dealing with Runtime Errors
  - Unit Testing
  - Profiling

- **REGULAR EXPRESSIONS:**
  - Python's Regular Expression Language:
    - Characters and Characters Classes
    - Quantifies, Grouping and Capturing
    - Assertions and flags
  - The Regular Expression Module

DEBUGGING, TESTING AND PROFILING

Debugging, Testing, and Profiling

- Writing programs is a mixture of art, craft, and science, and because it is done by humans, mistakes are made.

- Mistakes fall into several categories:

  - The quickest to reveal themselves and the easiest to fix are syntax errors, since these are usually due to typos.

  - More challenging are logical errors—with these, the program runs, but some aspect of its behavior is not what we intended or expected.
    - Many errors of this kind can be prevented from happening by using TDD (Test Driven Development),

  - Creating a program that has needlessly poor performance.
    - This is almost always due to a poor choice of algorithm or data structure or both.

Debugging

- Errors occurring in programming are called as bugs. The process of tracking this bugs is called as **debugging**.

- There are three types of errors that can occur while coding :
  - Syntax Error
  - Runtime Error and
  - Semantic Error.

- Making regular backups is an essential part of programming.

- Version control systems allows to incrementally save changes at whatever level of granularity is needed—every single change, or every set of related changes, or simply every so many minutes' worth of work.

- Many good cross-platform open source version control systems available are :
  - Bazaar (bazaar-vcs.org)
  - Mercurial (mercurial.selenic.com)
  - Git (git-scm.com) and
  - Subversion (subversion.tigris.org)

Debugging

- Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:
  - **Syntax errors** are produced by Python when it is translating the source code . They usually indicate that there is something wrong with the syntax of the program.
    - Example: Omitting the colon at the end of a def statement yields the somewhat redundant message SyntaxError: invalid syntax.
  - **Runtime errors** are produced by the runtime system if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing.
    - Example: An infinite recursion eventually causes a runtime error of maximum recursion depth exceeded.
  - **Semantic errors** are problems with a program that compiles and runs but doesn't do the right thing.
    - Example: An expression may not be evaluated in the order you expect, yielding an unexpected result.
- The first step in debugging is to figure out which kind of error you are dealing with.

Dealing with Syntax Errors

- The syntax is a defined structure or set of rules while writing a program.
- If someone fails to maintain correct syntax ,then it may lead to Syntax Error.
- Syntax errors are usually easy to fix once you figure out what they are.
- Unfortunately, the error messages are often not helpful.
  - The most common messages are SyntaxError: invalid syntax and SyntaxError: invalid token, neither of which is very informative.
- On the other hand, the message does tell you where in the program the problem occurred.
  - Actually, it tells you where Python noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

Syntax Error : Example

```
>>> a=1+2
>>> a
3

>>> 1+2=a
SyntaxError: can't assign to operator

>>> print(Hello, World!)
SyntaxError: invalid syntax
```

Dealing with Syntax Errors

- Here are some ways to avoid the most common syntax errors:
  - Make sure not use a Python keyword for a variable name.
  - Check to have a colon at the end of the header of every compound statement, including for, while, if, and def statements.
  - Check that indentation is consistent.
    - Indent with either spaces or tabs but it's best not to mix them. Each level should be nested the same amount.
  - Make sure that any strings in the code have matching quotation marks.
  - If using multiline strings with triple quotes (single or double), make sure to terminate the string properly.
    - An unterminated string may cause an invalid token error at the end of program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
  - An unclosed bracket – (, {, or [– makes Python continue with the next line as part of the current statement.
    - Generally, an error occurs almost immediately in the next line.
  - Check for the classic = instead of == inside a conditional.

Exercise

- Fix the syntax error in the following program, so that it prints out the sum of all the numbers from 1 to 10.

- Change at most one character.

```
print(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 +)
```

Dealing with Runtime Errors

- Runtime errors occur when the program executes.
- Since Python is an interpreted language, these errors will not occur until the flow of control in the program reaches the line with the problem.
- A **syntax error** happens when Python can't understand what you are saying.
- A **run-time error** happens when Python understands what you are saying, but runs into trouble when following the instructions.
- Examples:
  - using an undefined variable or function. This can also occur if you use capital letters inconsistently in a variable name:

```
callMe = "Maybe"
print(callme)
```

  - dividing by zero

```
print(1/0)
```

  - using operators on the wrong type of data

```
print("you cannot add text and numbers" + 12)
```

Runtime Error : Example

```
#runtime.py
a=2
b=3
c=4
d=input("enter a number")
sum=a+b+c+d
print(sum)
```

OUTPUT:

```
enter a number 5
Traceback (most recent call last): File
"C:/Python34/runtime error.py", line 5, in
sum=a+b+c+d
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Exercise

- Fix the run-time error in the following program, so that it prints out Hello on the first line and Joe on the second line.

- Can change at most two characters.

```
print("Hello")
username = Joe
print(username)
```

Semantic Error

- Semantic errors are also called as logical errors.
- Program can get executed without any error without producing desired output. Such type of error is called Semantic error.
- These usually do not produce any error message, but instead cause your program to behave incorrectly.
- These types of errors can be tricky to track down.
- It is difficult to track such errors as it requires to go backward to look for output and find errors causing incorrect output.
- Semantic or logic errors are problems with the design of program.
- These errors are often caused by accidentally using one variable in a place where a different variable is intended, or by simply doing some math incorrectly.

Semantic Error : Example

```
#Program to find sum of two digits
num1=int(input("Enter first number"))
num2=int(input("Enter second number"))
add = num1 - num2
print("sum of two nums is ", add )
```

OUTPUT:
Enter first number 200
Enter second number 57
sum of two nums is 143

Exercise

- You are going shopping for bread and milk, but there is tax. You buy $2.00 of milk and $4.00 of bread, and the tax rate is 3%.

- Print out the total cost of your groceries.

```
breadPrice = 4.00
breadTax = 0.03 * breadPrice
milkPrice = 2.00
milkTax = 0.03 * milkPrice
print(breadTax + breadPrice + milkTax + breadPrice)
```

Scientific Debugging

- The best way to eliminate logical errors is to prevent them from occurring in the first place by using TDD (Test Driven Development).

- To be able to kill a bug we must be able to do the following.
    1. Reproduce the bug.
    2. Locate the bug.
    3. Fix the bug.
    4. Test the fix.

- Scientific method of finding and fixing the bug:
    1. Think up an explanation—a hypothesis—that reasonably accounts for the bug.
    2. Create an experiment to test the hypothesis.
    3. Run the experiment.

Scientific Debugging

- There are two ways to instrument a program
  - inserting print() statements or
  - using a debugger.

- Both approaches are used to achieve the same end and both are valid, but some programmers have a strong preference for one or the other.

debug code

```
fv =int(input("Enter the amount to be received in the future: "))
r = float(input("Enter the rate of return (e.g. 0.05 for 5 percent): "))
n = int(input("Enter the number of years: "))

# calculate the pvfactor = 1 / (1+r)^n
pvfactor = 1 / (1+r) * n
# calcualte the pv = fv * pvfactor
pv = fv * pvfactor
# output the present value
print("That's worth ", pv, "today.")
```

Output:

```
Enter the amount to be received in the future: 100

Enter the rate of return (e.g. 0.05 for 5 percent): 0.05

Enter the number of years: 10
That's worth  952.3809523809524 today.
```

Using print to **debug code**

```
fv =int(input("Enter the amount to be received in the future: "))
r = float(input("Enter the rate of return (e.g. 0.05 for 5 percent): "))
n = int(input("Enter the number of years: "))
print("fv =", fv, "r = ", r, "n=", n  )

# calculate the pvfactor = 1 / (1+r)^n
pvfactor = 1 / (1+r) * n
print("pvfactor = ", pvfactor)
# calcualte the pv = fv * pvfactor
pv = fv * pvfactor
# output the present value
print("That's worth ", pv, "today.")
```

Output:

```
Enter the amount to be received in the future: 100

Enter the rate of return (e.g. 0.05 for 5 percent): 0.05

Enter the number of years: 10
fv = 100 r =  0.05 n= 10
pvfactor =  9.523809523809524
That's worth  952.3809523809524 today.
```

Using print to **debug code**

```
fv =int(input("Enter the amount to be received in the future: "))
r = float(input("Enter the rate of return (e.g. 0.05 for 5 percent): "))
n = int(input("Enter the number of years: "))
print("fv =", fv, "r = ", r, "n=", n  )

# calculate the pvfactor = 1 / (1+r)^n
pvfactor = 1 / (1+r) ** n
print("pvfactor = ", pvfactor)
# calcualte the pv = fv * pvfactor
pv = fv * pvfactor
# output the present value
print("That's worth ", pv, "today.")
```

Output:

```
Enter the amount to be received in the future: 100

Enter the rate of return (e.g. 0.05 for 5 percent): 0.05

Enter the number of years: 10
fv = 100 r =  0.05 n= 10
pvfactor =  0.6139132535407591
That's worth  61.39132535407591 today.
```

Using a debugger

- A debugger is a program that can helps to find out what is going on in a computer program.

- You can stop the execution at any prescribed line number, print out variables, continue execution, stop again, execute statements one by one, and repeat such actions until you have tracked down abnormal behavior and found bugs.

- Python has two standard debuggers.
  - One is supplied as a module (pdb - the python debugger), and can be used interactively in the console
    - Example: python3 -m pdb my_program.py
  - the easiest way to use it is to add import pdb in the program itself, and add the statement pdb.set_trace() as the first statement of the function to examine.
    - When the program is run, pdb stops it immediately after the pdb.set_trace() call, and allows us to step through the program, set breakpoints, and examine variables.

Debugging Python code using breakpoint()

- Debugging can be done using built-in function **breakpoint()** and **pdb module**.

- Python comes with the latest built-in function *breakpoint* which do the same thing as pdb.set_trace() .

- Introduce the breakpoint where you have doubt or somewhere you want to check for bugs or errors.

Debugger Example

```
def debugger(a, b):
    breakpoint()
    result = a / b
    return result

print(debugger(5, 0))
```

Output:

```
runfile('D:/Python/python code/brkpt.py',
wdir='D:/Python/python code')
> d:\python\python code\brkpt.py(11)debugger()
     9 def debugger(a, b):
    10     breakpoint()
---> 11     result = a / b
    12     return result
    13


ipdb > c
Traceback (most recent call last):

  File "D:\Python\python code\brkpt.py", line 14, in <module>
    print(debugger(5, 0))

  File "D:\Python\python code\brkpt.py", line 11, in debugger
    result = a / b

ZeroDivisionError: division by zero
```

Commands for debugging

- **c** -> continue execution
- **q** -> quit the debugger/execution
- **n** -> step to next line within the same function
- **s** -> step to next line in this function or a called function

Using a debugger

- The Python debugger comes as part of the standard Python distribution as a module called pdb.
- The module pdb defines an interactive source code debugger for Python programs.
- It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame.
- The debugger is also extensible, and is defined as the class pdb.

pdb module Example

```
def debugger(a, b):
    import pdb;
     pdb.set_trace()
    result = a / b
    return result

print(debugger(5, 0))
```

Output:

```
runfile('D:/Python/python code/untitled3.py',
wdir='D:/Python/python code')
> d:\python\python code\untitled3.py(11)debugger()


Ipdb > c
Traceback (most recent call last):

  File "D:\Python\python code\untitled3.py", line 14, in <module>
    print(debugger(5, 0))

  File "D:\Python\python code\untitled3.py", line 11, in debugger
    result = a / b

ZeroDivisionError: division by zero
```

pdb module Example

```
def debugger(a):
    import pdb;
     pdb.set_trace()
    result = [a[element] for element in range(0, len(a)+5)]
    return result

print(debugger([1, 2, 3]))
```

Output:

```
runfile('D:/Python/python code/untitled3.py', wdir='D:/Python/python code')
> d:\python\python code\untitled3.py(11)debugger()


ipdb > c
Traceback (most recent call last):

  File "D:\Python\python code\untitled3.py", line 14, in <module>
    print(debugger([1, 2, 3]))

  File "D:\Python\python code\untitled3.py", line 11, in debugger
    result = [a[element] for element in range(0, len(a)+5)]

  File "D:\Python\python code\untitled3.py", line 11, in <listcomp>
    result = [a[element] for element in range(0, len(a)+5)]

IndexError: list index out of range
```

Unit Testing

- Writing tests for the programs—if done well—can help reduce the incidence of bugs and can increase our confidence that the programs behave as expected.
- In general, testing cannot guarantee correctness.
- But by carefully choosing what we test we can improve the quality of our code.
- Unit testing is testing of the smallest possible pieces of a program.
- Unit testing is concerned with testing individual functions, classes, and methods, to ensure that they behave according to our expectations.

Unit Testing

- Python's standard library provides two unit testing modules:
  - doctest : Test specified in the documentation (i.e., in the docstring). It shows how to use the class and function to the user.
  - Unittest: Its in the Python Standard Library used to test code through separate file.

- In addition, there are third-party testing tools:
  - nose (code.google.com/p/python-nose) : more comprehensive and useful than the standard unittest module, while still being compatible with it.
  - py.test (codespeak.net/py/dist/test/test.html) : takes a somewhat different approach to unittest, and tries as much as possible to eliminate boilerplate test code.

Why testing?

- Square area function

```
def square_area(x):
    return x**3
```

Let's test this code!

Test the function, manually

```
>>> square_area(1)
1
```

So far so good, but...

```
>>> square_area(2)
8
```

The expected output is 4.

So, the program is incorrect when input = 2.

Then you fix your function

```
def square_area(x):
    return x**2
```

And you test again the function, manually

```
>>> square_area(1)
1
```

So far so good...

```
>>> square_area(2)
4
```

This was previously incorrect. Now we get it correct!

Systematic ways of testing

- doctest
- unittest

doctest

- `doctest` is a module provided by Python for module testing
- Comes prepackaged with Python
- lets you describe (and then run) good usage examples for your code
  - Simple, basically just have to include your test cases inside your function comment (= docstring)
- when code is changed, you can run those tests.
- also helps the reader/user to see what the designer's intention is
- from python.org:
  - The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown.

doctest modifies the docstring

- in the docstring at the top of the module, you can place special tests
- each test is preceeded by a >>>
- just after the test is the exact result of having run that test
- each test is run and the output compared. If no problems, then the module ‘passes’ and no output is produced

testmod

- once written, you must call the `doctest.testmod()` function

- doctest.testmod() will extract the code and the execution result from the docstrings (indicated with >>>), execute the extracted code and compare its result to the extracted result.

- It the output matches with the content, no output will be printed.

Square area with doctest

```
import doctest

def square_area(x):
    """
    compute the area of a square
    >>> square_area(1)
    1
    >>> square_area(2)
    4
    """
    return x**3

if __name__ == "__main__":
    doctest.testmod()
```

Square area with doctest : Output

```
runfile('D:/Python/python code/untitled0.py', wdir='D:/Python/python code')
**********************************************************************
File "D:\Python\python code\untitled0.py", line 15, in __main__.square_area
Failed example:
   square_area(2)
Expected:
   4
Got:
   8
**********************************************************************
1 items had failures:
  1 of   2 in __main__.square_area
***Test Failed*** 1 failures.
```

Square area with doctest : correct version

```
import doctest

def square_area(x):
    """
    compute the area of a square
    >>> square_area(1)
    1
    >>> square_area(2)
    4
    """
    return x**2

if __name__ == "__main__":
    doctest.testmod()
```

Square area with doctest : correct version output

When there is no error, there is nothing to print!

Exercise:

(1) Add a testcase for negative inputs, which should return nothing

(2) Fix the square area function to handle negative inputs

Square area with doctest handling negative inputs

```
import doctest

def square_area(x):
    """
    compute the area of a square
    >>> square_area(1)
    1
    >>> square_area(2)
    4
    >>> square_area(-1)
    """
    if x < 0:
        return None

    return x**2

if __name__ == "__main__":
    doctest.testmod()
```

Exercise

- Create a program and doctest to compute the area of a rectangle

Program and doctest to compute the area of a rectangle

```
import doctest

def rectangle_area(x,y):
    """
    compute the area of a rectangle
    >>> rectangle_area(1,1)
    1
    >>> rectangle_area(2,3)
    6
    >>> rectangle_area(-2,3)
    >>> rectangle_area(2,-3)
    >>> rectangle_area(100,200)
    20000
    """

    if x < 0 or y < 0:
        return None

    return x*y

if __name__ == "__main__":
    doctest.testmod()
```

unittest

- Comes prepackaged with Python
- unlike `doctest`, this is a separate file that describes in detail how to perform the overall test
- probably more useful, since it is not embedded in the sourcefile itself.
- As the name suggests, unit testing tests units or components of the code.

unittest features

- Every test class must be sub class of unittest.TestCase
- Every test function should start with test name.
- to check for an expected result use assert functions.
- The setUp() method define instructions that will be executed before test case.
- The tearDown() method define instructions that will be executed after test case.

Assert Methods

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

unittest for String methods

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])

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

unittest for String methods

- Output:

```
runfile('D:/Python/python code/untitled3.py', wdir='D:/Python/python code')
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

Square area with unittest

```
# square_oops.py

def square_area(x):
    return x**3
```

```
# square_oops_test.py

import unittest

from square_oops import square_area

class TestSquareArea(unittest.TestCase):

    def test_negative(self):
        self.assertEqual(square_area(-1), None)

    def test_positive(self):
        self.assertEqual(square_area(1),1)
        self.assertEqual(square_area(2),4)

    def test_positive_large(self):
        self.assertEqual(square_area(40),1600)

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

Square area with unittest

```
FAIL: test_negative (__main__.TestSquareArea)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Python\python code\square_oops_test.py", line 16, in test_negative
    self.assertEqual(square_area(-1), None)
AssertionError: -1 != None

======================================================================
FAIL: test_positive (__main__.TestSquareArea)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Python\python code\square_oops_test.py", line 20, in test_positive
    self.assertEqual(square_area(2),4)
AssertionError: 8 != 4

======================================================================
FAIL: test_positive_large (__main__.TestSquareArea)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Python\python code\square_oops_test.py", line 23, in test_positive_large
    self.assertEqual(square_area(40),1600)
AssertionError: 64000 != 1600

----------------------------------------------------------------------
Ran 3 tests in 0.004s

FAILED (failures=3)
```

Square area with unittest : correct version

```
# square.py

def square_area(x):
    if x < 0: return None
    return x**2
```

```
# square_test.py

import unittest

from square import square_area

class TestSquareArea(unittest.TestCase):

    def test_negative(self):
        self.assertEqual(square_area(-1), None)

    def test_positive(self):
        self.assertEqual(square_area(1),1)
        self.assertEqual(square_area(2),4)

    def test_positive_large(self):
        self.assertEqual(square_area(40),1600)

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

Square area with unittest : correct version

- Output:

```
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

Exercise

- Create a program and unittest to compute the area of a rectangle

Rectangle area with unittest

```
# rectangle.py

def rectangle_area(x,y):
    if x < 0 or y < 0: return None
    return x*y
```

```
# rectangle_test.py

import unittest

from rectangle import rectangle_area

class TestRectangleArea(unittest.TestCase):

    def test_negative(self):
        self.assertEqual(rectangle_area(-1,1), None)
        self.assertEqual(rectangle_area(1,-1), None)
        self.assertEqual(rectangle_area(-1,-1), None)

    def test_positive(self):
        self.assertEqual(rectangle_area(1,1),1)
        self.assertEqual(rectangle_area(2,3),6)

    def test_positive_large(self):
        self.assertEqual(rectangle_area(40,40),1600)
        self.assertEqual(rectangle_area(40,500),20000)

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

Rectangle area with unittest

- Output:

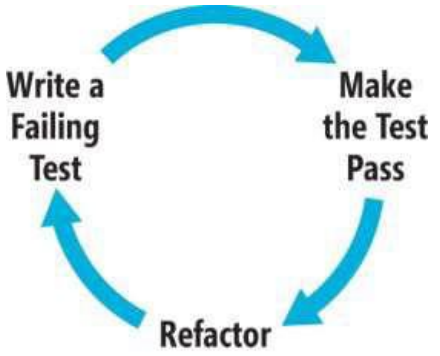
```
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

Test-Driven Development (TDD)

- A method of software development to define tests before actually starting coding!
- The tests define the desired behavior of the program.
- In this way, you code with the mission to satisfy the tests!
- The process of implementing code by writing your tests first, seeing them fail, then writing the code to make the tests pass.

TDD Stages

1. Before writing any code, write test cases.
2. Run the tests, this would surely fail since there is no code yet.
3. Make working code as minimum as possible to satisfy the tests.
4. Run the tests, and check if your code passes. If not, fix your code until you pass.
5. Now, time to refactor (= clean) your code, make sure the refactored code passes the tests.
6. Repeat 1-5 for other program features.



What is Profiling?

- Measuring the execution time
- Measuring Performance
- Insight of run time performance of a given piece of code
- Frequently used to optimize execution time
- Used to analyze other characteristics such as memory consumption

Why do we need Python Profilers ?

- The main aspect while writing any code, especially when deploying, is that it should consume the lowest computational time and cost.

- This is especially important when you run code on cloud services, where there is a defined cost associated with the usage of computing resources.

  - If there are two pieces of code that give the same result, the one that takes the least time and resource is usually chosen.

- **Python Profilers** can answer the question **Why does my code take so long to run?**.
  - **It tells which part of the code took how long to run**.
    - This lets you focus on that particular part and achieve efficiency.

Why do we need Python Profilers ? Contd..

- Profiler can be used to answer questions like :
  - Why is this program slow?
  - What is actually happening when this code executes?
  - Is there anything that I can improve?
  - How much memory is consumed by program?
  - How much time is taken by each function for execution?

Profiling

- Here are few Python programming habits that are easy to learn and apply, and that are good for performance.
- None of the techniques is Python-version-specific, and all of them are perfectly sound Python programming style.
  - First, prefer tuples to lists when a read–only sequence is needed.
  - Second, use generators rather than creating large tuples or lists to iterate over.
  - Third, use Python's built-in data structures—dicts, lists, and tuples—rather than custom data structures implemented in Python, since the built-in ones are all very highly optimized.
  - Fourth, when creating large strings out of lots of small strings, instead of concatenating the small strings, accumulate them all in a list, and join the list of strings into a single string at the end.
  - Fifth and finally, if an object (including a function or method) is accessed a large number of times using attribute access (e.g., when accessing a function in a module), or from a data structure, it may be better to create and use a local variable that refers to the object to provide faster access.

Profiling

- Python's standard library provides two modules that are particularly useful to investigate the performance of code.
  - **timeit module -** useful for timing small pieces of Python code, and can be used, for example, to compare the performance of two or more implementations of a particular function or method.
  - **cProfile module -** used to profile a program's performance where it provides a detailed breakdown of call counts and times and so can be used to find performance bottlenecks.

timeit()

- **timeit()** method is used to get the execution time taken for the small code given.
  - The library runs the code statement 1 million times and provides the minimum time taken from the set.

- It is a useful method that helps in checking the performance of the code.

- **Syntax:**
  - timeit.timeit(stmt, setup,timer, number)

    - **stmt**: This will take the code for which you want to measure the execution time. The default value is "pass".

    - **setup**: This will have setup details that need to be executed before stmt. The default value is "pass.“

    - **timer**: This will have the timer value, timeit() already has a default value set, and we can ignore it.

    - **number**: The stmt will execute as per the number is given here. The default value is 1000000.

timeit()

- To work with timeit(), import the module :
  import timeit

- **Example :**

```
# testing timeit()

import timeit

print(timeit.timeit('output = 10*5'))
```

Output:
0.0612788039999999

Timing Multiple lines in python code

- There are two ways to execute multiple lines of code in timeit.timeit():
  - using a semicolon or
  - saving the code enclosed as a string with triple quotes.

Example 1: Using semicolon

```
import timeit

print("The time taken is ",timeit.timeit(stmt='a=10;b=10;sum=a+b'))
```

Output:

The time taken is 0.137031482

Example 2: Using triple quotes

```
import timeit
import_module = "import random"
testcode = '''
def test():
   return random.randint(10, 100)
'''
print(timeit.repeat(stmt=testcode, setup=import_module))
```

Output:

[0.09283898700050486, 0.0921019580000575, 0.0917603859998053, 0.09019299699957628, 0.08942999000009877]

timeit - Methods

- Here, are 2 important timeit methods:
  - **timeit.default_timer()** : This will return the default time when executed.
  - **timeit.repeat(stmt, setup, timer, repeat, number)** : same as timeit() , but with repeat the timeit() is called the number of times repeat is given.

default_timer() Example

```
import timeit
import random

def test():
    return random.randint(10, 100)

starttime = timeit.default_timer()
print("The start time is :",starttime)
test()
print("The time difference is :", timeit.default_timer() - starttime)
```

Output:

The start time is : 5923.030127087

The time difference is : 4.45809992015711e-05

Example : timeit.repeat()

Note:
timeit.repeat() works similar to timeit.timeit() function, with the only difference it takes in the repeat argument and gives back the execution time in array format with values as per the repeat number.

```
# testing timeit()
import timeit
import_module = "import random"
testcode = '''
def test():
    return random.randint(10, 100)
'''
print(timeit.repeat(stmt=testcode, setup=import_module, repeat=5))
```

Output:

```
[0.09730092900008458, 0.09672426399993128, 0.09795745500014164,
    0.09709694699995453, 0.09572872999979154]
```

timeit Module

- **Advantages**
  - Easy to use
  - Simple to understand
  - Measure execution time of small code snippets

- **Disadvantages**
  - Simple code only
  - Not very deterministic
  - Have to manually create runnable code snippets
  - Manual analysis

Why use timeit()?

- Here are a few reasons why timeit() is the best way to measure execution time.
  - It runs the code statement 1 million times that is the default value, and from that, it will return you the minimum time taken.
    - You can also increase/decrease the 1 million by setting the argument number in time () function.
  - While executing the test, the garbage collection is disabled every time by time () function.
  - timeit() internally takes the accurate time as per your operating system being used.
    - For example, it will use time.clock() for Windows operating system and time.time() for mac and Linux.

Introduction to cProfile

- cProfile is a built-in python module that can perform profiling. It is the most commonly used profiler currently.

- But, why cProfile is preferred?

  - It gives you the **total run time taken by the entire code**.
  - It also shows **the time taken by each individual step**. This allows you to compare and find which parts need optimization
  - cProfile module also tells the **number of times certain functions are being called**.
  - The data inferred can be **exported easily** using pstats module.
  - The data can be **visualized** nicely using snakeviz module.

How to use cProfile ?

- cProfile provides a simple **run() function which is sufficient for most cases**.

- Syntax :

  - cProfile.run(statement, filename=None, sort=-1)

    - **Pass python code or a function name that you want to profile as a string** to the statement argument.

    - If you want to save the output in a file, it can be passed to the filename argument.

    - The sort argument can be used to specify how the output has to be printed. By default, it is set to -1(no value).

Example

```
import cProfile

cProfile.run('2 + 2')
```

Ouput:

```
3 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Example

- Line no.1: shows the number of function calls and the time it took to run.
- Line no.2: Ordered by: standard name means that the text string in the far right column was used to sort the output. This could be changed by the sort parameter.
- Line no. 3 onwards contain the functions and sub functions called internally. Let's see what each column in the table means.
  - **ncalls**: The number of times the function was called.
  - **tottime**: The total time spent in the function without taking into account the calls to other functions.
  - **cumtime**: The time in the function including other function calls.
  - **percall**: The time spent for a single call of the function--it can be obtained by dividing the total or cumulative time by the number of calls.
  - **filename:lineno**: The filename and corresponding line numbers. This information is not available when calling C extensions modules.

Profiling a function that calls other functions

```
import cProfile

# Code containing multiple dunctions
def create_array():
  arr=[]
  for i in range(0,400):
    arr.append(i)

def print_statement():
  print('Array created successfully')


def main():
  create_array()
  print_statement()


if __name__ == '__main__':
    cProfile.run('main()')
```

Output:

```
Array created successfully
         500041 function calls in 0.138 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.138    0.138 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 cprofile_demo.py:10(print_statement)
        1    0.006    0.006    0.138    0.138 cprofile_demo.py:14(main)
        1    0.075    0.075    0.132    0.132 cprofile_demo.py:5(create_array)
        3    0.000    0.000    0.000    0.000 iostream.py:197(schedule)
        2    0.000    0.000    0.000    0.000 iostream.py:309(_is_master_process)
        2    0.000    0.000    0.000    0.000 iostream.py:322(_schedule_flush)
        2    0.000    0.000    0.000    0.000 iostream.py:384(write)
        3    0.000    0.000    0.000    0.000 iostream.py:93(_event_pipe)
        3    0.000    0.000    0.000    0.000 socket.py:342(send)
        3    0.000    0.000    0.000    0.000 threading.py:1050(_wait_for_tstate_lock)
        3    0.000    0.000    0.000    0.000 threading.py:1092(is_alive)
        3    0.000    0.000    0.000    0.000 threading.py:507(is_set)
        1    0.000    0.000    0.138    0.138 {built-in method builtins.exec}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        2    0.000    0.000    0.000    0.000 {built-in method nt.getpid}
        3    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' objects}
        3    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
   500000    0.056    0.000    0.056    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Profiling a function that calls other functions

- Notice that when a particular function is called more than once, the ncalls value reflects that.

- Also spot the difference between the tottime and cumtime. This output clearly tells you that for i in range(0,500000) is the part where majority of time is spent.

How to use Profile class of cProfile

- Even though the run() function of cProfile may be enough in some cases, there are certain other methods that are useful as well.
- The Profile() class of cProfile gives you more precise control.
- By default, cProfile sorts its output by “standard name”. This means that it sorts by the filename(far right column).

How to use Profile to modify reports?

- To find the time-consuming parts, it would be helpful to sort the outputs as per ncalls.

- To do this:
  - First, initialize an instance of Profile class.
  - After that, call the enable() method of the profiler to start collecting profiling data.
  - After that, call the function you want to profile.
  - To stop collecting profiling data, call the disable() method.

How to report the data collected ?

- The pstats module can be used to manipulate the results collected by the profiler object.
  - First, create an instance of the stats class using pstats.Stats.
  - Next, use the Stats class to create a statistics object from a profile object through stats= pstats.Stats(profiler).
  - Now, to sort the output by ncalls, use the sort_stats() method.
  - Finally to print the output, call the function print_statss() of stats object.

```
# How to use Profile class of cProfile

def create_array():
  arr=[]
  for i in range(0,400000):
    arr.append(i)

def print_statement():
  print('Array created successfully')


def main():
  create_array()
  print_statement()

if __name__ == '__main__':
    import cProfile, pstats
    profiler = cProfile.Profile()
    profiler.enable()
    main()
    profiler.disable()
    stats = pstats.Stats(profiler).sort_stats('ncalls')
    stats.print_stats()
```

Output:

```
Array created successfully
         400039 function calls in 0.110 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   400000    0.041    0.000    0.041    0.000 {method 'append' of 'list' objects}
        3    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\threading.py:507(is_set)
        3    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\threading.py:1050(_wait_for_tstate_lock)
        3    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\threading.py:1092(is_alive)
        3    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\zmq\sugar\socket.py:342(send)
        3    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:93(_event_pipe)
        3    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:197(schedule)
        3    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
        3    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' objects}
        2    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:309(_is_master_process)
        2    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:322(_schedule_flush)
        2    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:384(write)
        2    0.000    0.000    0.000    0.000 {built-in method nt.getpid}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        1    0.064    0.064    0.105    0.105 D:\Python\python code\profile_class_demo.py:2(create_array)
        1    0.000    0.000    0.000    0.000 D:\Python\python code\profile_class_demo.py:7(print_statement)
        1    0.005    0.005    0.110    0.110 D:\Python\python code\profile_class_demo.py:11(main)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

To sort the output by the cumulative time

```
# Sort output by Cumulative time
if __name__ == '__main__':
    import cProfile, pstats
    profiler = cProfile.Profile()
    profiler.enable()
    main()
    profiler.disable()
    stats = pstats.Stats(profiler).sort_stats('cumtime')
    stats.print_stats()
```

Output:

Array created successfully

400039 function calls in 0.087 seconds

Ordered by: cumulative time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.005	0.005	0.087	0.087	D:\Python\python code\profile_class_demo.py:11(main)
1	0.052	0.052	0.082	0.082	D:\Python\python code\profile_class_demo.py:2(create_array)
400000	0.029	0.000	0.029	0.000	{method 'append' of 'list' objects}
1	0.000	0.000	0.000	0.000	D:\Python\python code\profile_class_demo.py:7(print_statement)
1	0.000	0.000	0.000	0.000	{built-in method builtins.print}
2	0.000	0.000	0.000	0.000	C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:384(write)
3	0.000	0.000	0.000	0.000	C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:197(schedule)
3	0.000	0.000	0.000	0.000	C:\ProgramData\Anaconda3\lib\site-packages\zmq\sugar\socket.py:342(send)
3	0.000	0.000	0.000	0.000	C:\ProgramData\Anaconda3\lib\threading.py:1092(is_alive)
2	0.000	0.000	0.000	0.000	C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:322(_schedule_flush)
3	0.000	0.000	0.000	0.000	C:\ProgramData\Anaconda3\lib\threading.py:1050(_wait_for_tstate_lock)
3	0.000	0.000	0.000	0.000	{method 'acquire' of '_thread.lock' objects}
2	0.000	0.000	0.000	0.000	C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:309(_is_master_process)
3	0.000	0.000	0.000	0.000	C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:93(_event_pipe)
2	0.000	0.000	0.000	0.000	{built-in method nt.getpid}
3	0.000	0.000	0.000	0.000	{method 'append' of 'collections.deque' objects}
2	0.000	0.000	0.000	0.000	{built-in method builtins.isinstance}
3	0.000	0.000	0.000	0.000	C:\ProgramData\Anaconda3\lib\threading.py:507(is_set)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

How to export cProfile data?

- The pstats module comes to use here.
- Create a Stats instance and pass the profiler as input to it as shown below.
- After that, use dump_stats() method to store it to any file by providing the path.
- **# Export profiler output to file**

```
stats = pstats.Stats(profiler)
stats.dump_stats('/content/export-data')
```

```
import random

def print_msg():
    for i in range(10):
        print("Program completed")

def generate():
    data = [random.randint(0, 99) for p in range(0, 100)]
    return data

def search_function(data):
    for i in data:
        if i in [100,200,300,400,500]:
            print("success")

def main():
    data=generate()
    search_function(data)
    print_msg()
```

```
if __name__ == '__main__':
    import cProfile, pstats
    profiler = cProfile.Profile()
    profiler.enable()
    main()
    profiler.disable()
    stats =
pstats.Stats(profiler).sort_stats('tottime')
    stats.print_stats()
```

Output:

Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed

811 function calls in 0.001 seconds

```
Ordered by: internal time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      21    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\zmq\sugar\socket.py:342(send)
     100    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\random.py:174(randrange)
     100    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\random.py:224(_randbelow)
      20    0.000    0.000    0.001    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:384(write)
      21    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:197(schedule)
     100    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\random.py:218(randint)
       1    0.000    0.000    0.000    0.000 D:\Python\python code\stats_demo.py:9(<listcomp>)
      21    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\threading.py:1092(is_alive)
     127    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random' objects}
      10    0.000    0.000    0.001    0.000 {built-in method builtins.print}
      20    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:309(_is_master_process)
       1    0.000    0.000    0.000    0.000 D:\Python\python code\stats_demo.py:12(search_function)
      21    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\threading.py:1050(_wait_for_tstate_lock)
      21    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' objects}
      21    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:93(_event_pipe)
     100    0.000    0.000    0.000    0.000 {method 'bit_length' of 'int' objects}
      20    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\site-packages\ipykernel\iostream.py:322(_schedule_flush)
       1    0.000    0.000    0.001    0.001 D:\Python\python code\stats_demo.py:4(print_msg)
      21    0.000    0.000    0.000    0.000 C:\ProgramData\Anaconda3\lib\threading.py:507(is_set)
      21    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
      20    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
       1    0.000    0.000    0.000    0.000 D:\Python\python code\stats_demo.py:8(generate)
      20    0.000    0.000    0.000    0.000 {built-in method nt.getpid}
       1    0.000    0.000    0.001    0.001 D:\Python\python code\stats_demo.py:17(main)
       1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

- The pstats module provides the function strip_dirs() to overcome the output that is cluttered and difficult to read.

- It removes all leading path information from file names.

- **# Remove dir names**

  stats**.**strip_dirs**()**

  stats**.**print_stats**()**

```
# Using cProfile.Profile example
import random
def print_msg():
    for i in range(10):
        print("Program completed")
def generate():
    data = [random.randint(0, 99) for p in range(0, 100)]
    return data
def search_function(data):
    for i in data:
        if i in [100,200,300,400,500]:
            print("success")
def main():
    data=generate()
    search_function(data)
    print_msg()

if __name__ == '__main__':
    import cProfile, pstats
    profiler = cProfile.Profile()
    profiler.enable()
    main()
    profiler.disable()
    stats = pstats.Stats(profiler).sort_stats('tottime')
    #stats.print_stats()
    # Remove dir names
    stats.strip_dirs()
    stats.print_stats()
```

Output:

```
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
        810 function calls in 0.001 seconds
```

Output:

Random listing order was used

```
  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      21    0.000    0.000    0.000    0.000 threading.py:507(is_set)
      21    0.000    0.000    0.000    0.000 threading.py:1050(_wait_for_tstate_lock)
      21    0.000    0.000    0.000    0.000 threading.py:1092(is_alive)
     100    0.000    0.000    0.000    0.000 random.py:174(randrange)
     100    0.000    0.000    0.000    0.000 random.py:218(randint)
     100    0.000    0.000    0.000    0.000 random.py:224(_randbelow)
      21    0.000    0.000    0.000    0.000 socket.py:342(send)
      21    0.000    0.000    0.000    0.000 iostream.py:93(_event_pipe)
      21    0.000    0.000    0.000    0.000 iostream.py:197(schedule)
      20    0.000    0.000    0.000    0.000 iostream.py:309(_is_master_process)
      20    0.000    0.000    0.000    0.000 iostream.py:322(_schedule_flush)
      20    0.000    0.000    0.000    0.000 iostream.py:384(write)
       1    0.000    0.000    0.000    0.000 stats_demo.py:9(<listcomp>)
       1    0.000    0.000    0.000    0.000 stats_demo.py:4(print_msg)
       1    0.000    0.000    0.000    0.000 stats_demo.py:12(search_function)
       1    0.000    0.000    0.001    0.001 stats_demo.py:17(main)
       1    0.000    0.000    0.000    0.000 stats_demo.py:8(generate)
      21    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
      20    0.000    0.000    0.000    0.000 {built-in method nt.getpid}
      21    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' objects}
     100    0.000    0.000    0.000    0.000 {method 'bit_length' of 'int' objects}
      20    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
      10    0.000    0.000    0.000    0.000 {built-in method builtins.print}
       1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     126    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random' objects}
```

How to visualize cProfile reports?

- A good solution to get a clear picture of the profiling data is to visualize it.
- A best tool available at the moment for visualizing data obtained by cProfile module is SnakeViz.
- SnakeViz is a browser based graphical viewer for the output of Python's cProfile module
- For Ipython notebooks like google colab and Jupyter, load the SnakViz extension using %load_ext snakeviz command.
  - (Installation - !pip install snakeviz)
- After this, call the function or program’s profiling you want to visualize through the %snakeviz <filename>.
  - The filename can be either the entire python script or call to a particular function.

SnakeViz

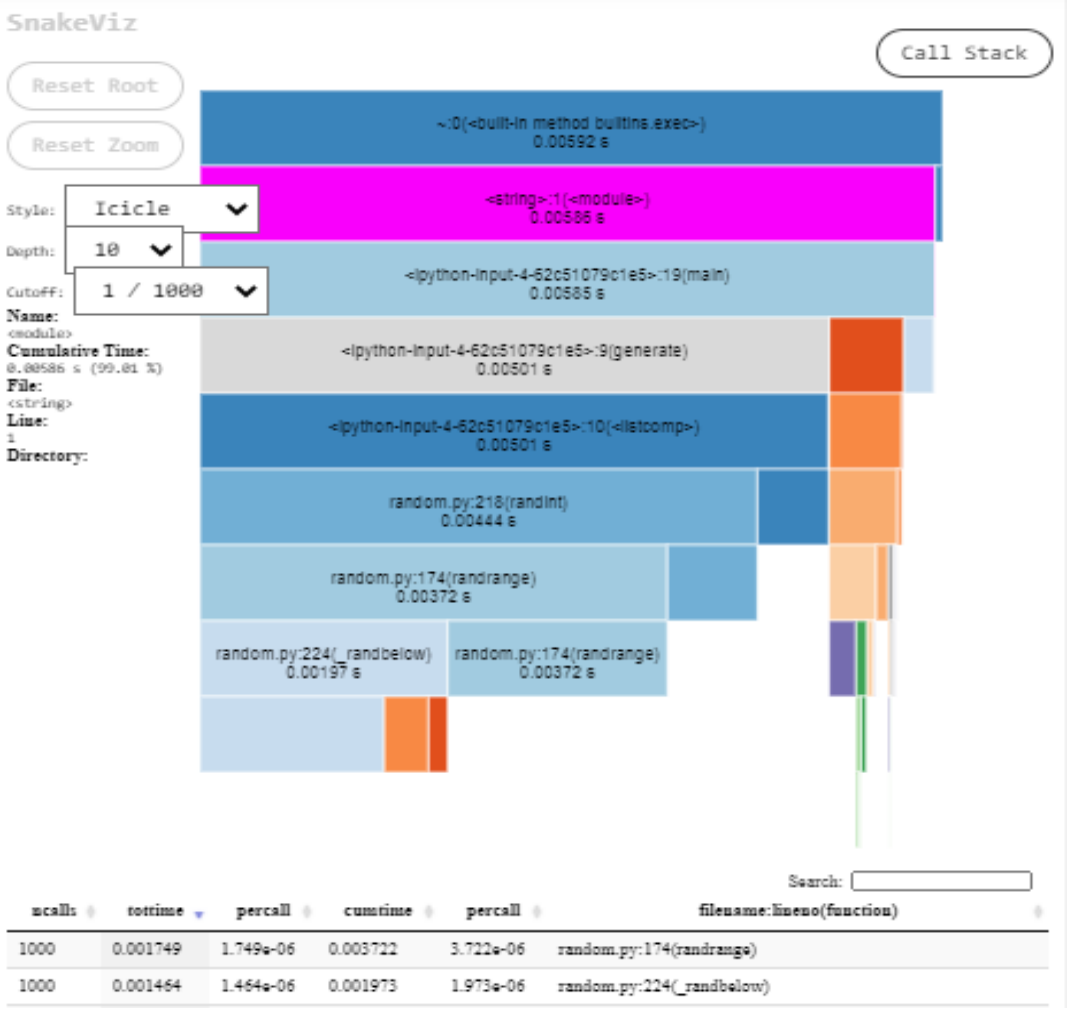
```
# Code to test visualization
import random
# Simple function to print messages
def print_msg():
    for i in range(10):
        print("Program completed")

# Generate random data
def generate():
    data = [random.randint(0, 99) for p in range(0, 1000)]
    return data

# Function to search
def search_function(data):
    for i in data:
        if i in [100,200,300,400,500]:
            print("success")

def main():
    data=generate()
    search_function(data)
    print_msg()


%load_ext snakeviz
%snakeviz main()
```

Output:

```
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
Program completed
```

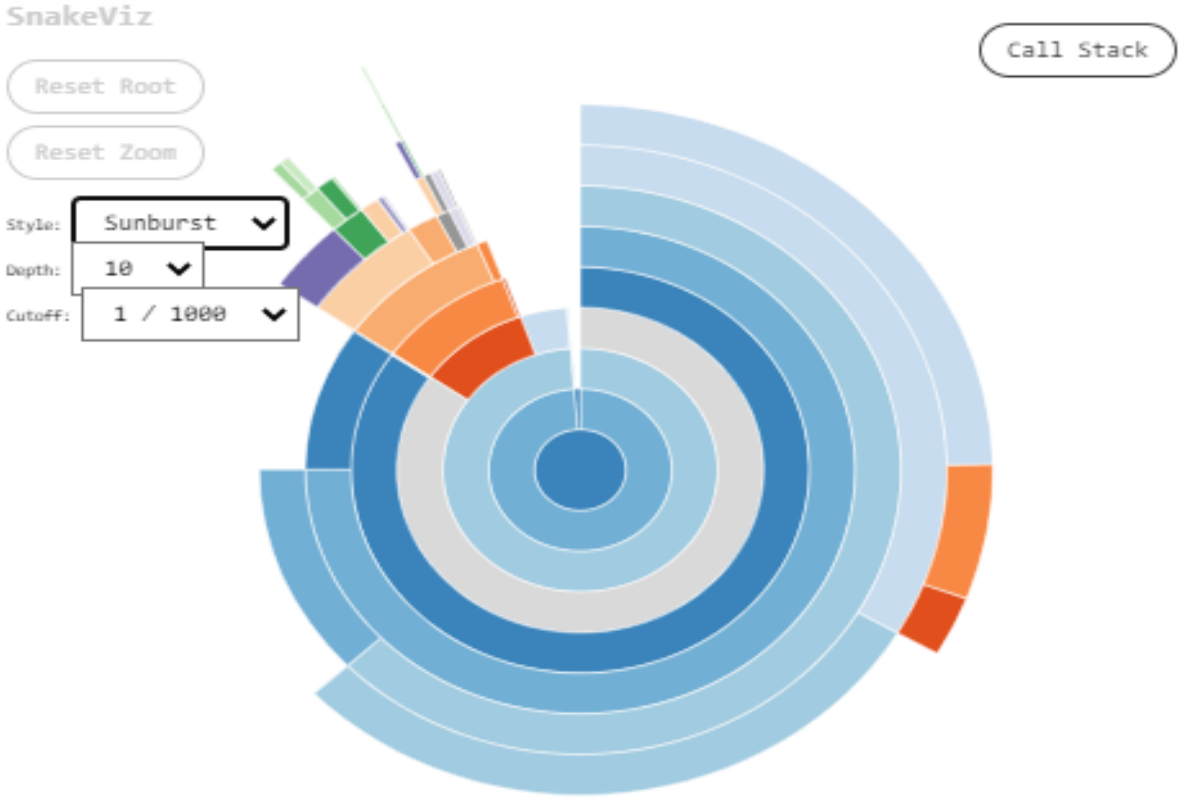
*** Profile stats marshalled to file 'C:\\Users\\ISEDSCE\\AppData\\Local\\Temp\\tmpv6guj4dx'.

Embedding SnakeViz in this document...



SnakeViz

- SnakeViz has two visualization styles, ‘icicle’ and ‘sunburst’.
- By default, it’s icicle. icicle, the fraction of time taken by a code is represented by the width of the rectangle.
- In Sunburst, it is represented by the angular extent of an arc.
- One can switch between the two styles using the “Style” dropdown.
- For the same code, the Sunburst style visualization



Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000	0.001749	1.749e-06	0.003722	3.722e-06	random.py:174(randrange)
1000	0.001464	1.464e-06	0.001973	1.973e-06	random.py:224(_randbelow)
1000	0.0007168	7.168e-07	0.004439	4.439e-06	random.py:218(randint)

REGULAR EXPRESSIONS

Regular Expressions

- A regular expression is a compact notation for representing a collection of strings.

- Regular expressions (“regexes”) are defined using a mini-language that is completely different from Python—but Python includes the re module to create and use regexes.

Regexes

- Regexes are used for five main purposes:
  - **Parsing**: identifying and extracting pieces of text that match certain criteria—regexes are used for creating ad hoc parsers and also by traditional parsing tools
  - **Searching**: locating substrings that can have more than one form, for example, finding any of “pet.png”, “pet.jpg”, “pet.jpeg”, or “pet.svg” while avoiding “carpet.png” and similar
  - **Searching and replacing**: replacing everywhere the regex matches with a string, for example, finding “bicycle” or “human powered vehicle” and replacing either with “bike”
  - **Splitting strings**: splitting a string at each place the regex matches, for example, splitting everywhere colon-space or equals (“: ” or “=”) occurs
  - **Validation**: checking whether a piece of text meets some criteria, for example, contains a currency symbol followed by digits

Regexes

- The regexes are widely used for searching, splitting, and validation.

- Regexes used to create parsers have a limitation :
  - They are only able to deal with recursively structured text if the maximum level of recursion is known.
  - large and complex regexes can be difficult to read and maintain.

- For parsing the best approach is to use a tool designed for the purpose—for example, use a dedicated XML parser for XML

Python - Regex

- A regular expression also known as regex is a sequence of characters that defines a search pattern.
- Regular expressions are used in search algorithms, search and replace dialogs of text editors, and in lexical analysis. It is also used for input validation.
- Regular expressions use two types of characters in the matching pattern string:
  - Meta characters :characters having a special meaning, similar to * in wild card. (Example: . ^ $ * + ? { } [ ] \ | ( ))
  - Literals are alphanumeric characters.
- The most common applications of regular expressions are:
  - Search for a pattern in a string
  - Finding a pattern string
  - Break string into a sub strings
  - Replace part of a string

Raw strings

- Methods in re module use raw strings as the pattern argument.
  - A raw string is having prefix 'r' or 'R' to the normal string literal.

```
>>> normal="computer"
>>> print (normal)
computer
>>> raw=r"computer"
>>> print (raw)
Computer
```

- Both strings appear similar. The difference is evident when the string literal embeds escape characters ('\n', '\t' etc.)

```
>>> normal="Hello\nWorld"
>>> print (normal)
Hello World
>>> raw=r"Hello\nWorld"
>>> print (raw)
Hello\nWorld
```

Character Classes or Character Sets

- The square brackets[and] are used for specifying a set of characters that you wish to match.

- Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-'.

[abc]	Match any of the characters a, b, or c
[a-c]	Which uses a range to express the same set of characters.
[a-z]	Match only lowercase letters.
[0-9]	Match only digits.
'^'	Complements the character set in [].[^5] will match any character except '5'.

Character Class Shorthands

Symbol	Meaning
.	Matches any character except newline; or any character at all with the re.DOTALL flag; or inside a character class matches a literal .
\d	Matches a Unicode digit; or [0-9] with the re.ASCII flag
\D	Matches a Unicode nondigit; or [^0-9] with the re.ASCII flag
\s	Matches a Unicode whitespace; or [\t\n\r\f\v] with the re.ASCII flag
\S	Matches a Unicode nonwhitespace; or [^ \t\n\r\f\v] with the re.ASCII flag
\w	Matches a Unicode "word" character; or [a-zA-Z0-9_] with the re.ASCII flag
\W	Matches a Unicode non-"word" character; or [^a-zA-Z0-9_] with the re.ASCII flag

Character Classes or Character Sets : Examples

- gr[ae]y - matches either gray or grey.
- [0-9a-fA-F] matches a single hexadecimal digit, case insensitively.
- [A-Za-z_][A-Za-z_0-9]* finds an identifier in a programming language.
- [^0-9\r\n] matches any character that is not a digit or a line break.
- [\\x] matches a backslash or an x
- [x^] matches an x or a caret.
  - To include an unescaped caret as a literal, place it anywhere except right after the opening bracket.
- []x] matches a closing bracket or an x.
- [^]x] matches any character that is not a closing bracket or an x.
  - An un-escaped closing bracket can be included by placing it right after the opening bracket, or right after the negating caret.
- [0-9]+ can match 837 as well as 222
  - Repeat a character class by using the ?, * or + operators
- ([0-9])\1+ matches 222 but not 837.
  - To repeat the matched character, rather than the class, use backreferences

Quantifiers

- A quantifier has the form ***{m,n} where m and n are the minimum and maximum*** times the expression the quantifier applies to must match.
  - Example : both **e{1,1}e{1,1} and e{2,2} match feel, but neither matches felt.**
- The regex language supports several convenient shorthands. If only one number is given in the quantifier it is taken to be both the minimum and the maximum.
  - Ex: **e{2} is the same as e{2,2}. I**f no quantifier is explicitly given, it is assumed to be one (i.e., **{1,1} or {1}); therefore, ee is the same as e{1,1}e{1,1}** and **e{1}e{1}.**
- Having a different minimum and maximum is often convenient.
  - For example, to match travelled and traveled (both legitimate spellings), use either **travel{1,2}ed or travell{0,1}ed.**
    - **The {0,1} quantification is so often used that** it has its own shorthand form, **?, so another way of writing the regex (and the** one most likely to be used in practice) is **travell?ed.**

Quantifiers (contd..)

- Two other quantification shorthands are provided:
  - **+ which stands for {1,*n*} (“*at* least one”)**
    - **The + quantifier is very useful.**
    - **Example: to match integers use \d+ since this matches one or more digits.**
      - **This regex could match in two places in the string 4588.91, for example, 4588.91 and 4588.91.**
  - *** which stands for {0,*n*} (“*any number of*”);**
    - The *** quantifier is less useful, simply because it can so often lead to unexpected** results.
    - ***Example:* #*** will match any line whatsoever, including blank lines because the meaning is “match any number of #s”—and that includes none.

***n is the* maximum possible number allowed for a quantifier, usually at least 32767.**

- ***Note : As a rule of thumb for avoid using * at all, and if you use it (or if you use ?), make sure there is at least one other expression in the regex that has a nonzero quantifier***

Regular Expression Quantifiers

Syntax	Meaning
e? or *e*{0,1}	Greedily match zero or one occurrence of expression *e*
e?? or *e*{0,1}?	Nongreedily match zero or one occurrence of expression *e*
e+ or *e*{1,}	Greedily match one or more occurrences of expression *e*
e+? or *e*{1,}?	Nongreedily match one or more occurrences of expression *e*
*e** or *e*{0,}	Greedily match zero or more occurrences of expression *e*
*e**? or *e*{0,}?	Nongreedily match zero or more occurrences of expression *e*
e{*m*}	Match exactly *m* occurrences of expression *e*
e{*m*,}	Greedily match at least *m* occurrences of expression *e*
e{*m*,}?	Nongreedily match at least *m* occurrences of expression *e*
e{,*n*}	Greedily match at most *n* occurrences of expression *e*
e{,*n*}?	Nongreedily match at most *n* occurrences of expression *e*
e{*m*,*n*}	Greedily match at least *m* and at most *n* occurrences of expression *e*
e{*m*,*n*}?	Nongreedily match at least *m* and at most *n* occurrences of expression *e*

Quantifiers (contd..)

- By default, all quantifiers are *greedy—they* match as many characters as they can.

- Quantifier can be made nongreedy (also called *minimal) by following it with a **? symbol.***
  - The question mark has two different meanings :
    - on its own it is a shorthand for the **{0,1} quantifier,** and
    - when it follows a quantifier it tells the quantifier to be nongreedy
      - Example: **\d+ it will match 136. \d+? can match the string 136 in three different places: 136, 136, and** 136.

- Nongreedy quantifiers can be useful for quick XML and HTML parsing.

Grouping and Capturing

- Regexes can match any one of two or more alternatives, capture the match or some part of the match for further processing. Also, a quantifier can sometimes be used to apply several expressions.

- All of these can be achieved by grouping with (), and in the case of alternatives using alternation with |.

- Alternation is especially useful to match any one of several quite different alternatives.

  - Example: the regex **aircraft|airplane|jet or air(craft|plane)|jet** will match any text that contains “aircraft” or “airplane” or “jet”.

- Parentheses serve two different purposes—to group expressions and to capture the text that matches an expression.

Grouping and Capturing (contd..)

- Group refers to a grouped expression whether it captures or not
- Capture and capture group refesr to a captured group.
- A grouped expression is an expression and so can be quantified.
- Example: to read a text file with lines of the form key=value, where each key is alphanumeric, the regex **(\w+)=(.+) will match every line that has a** nonempty key and a nonempty value. For every line that matches, two captures are made, the first being the key and the second being the value.

Grouping and Capturing (contd..)

- Captures can be referred to using *backreferences, that is, by referring back to* an earlier capture group.
- One syntax for backreferences inside regexes themselves is ***\i where i is the capture number.***
- ***Captures are numbered starting*** from one and increasing by one going from left to right as each new (capturing) left parenthesis is encountered.
- Example: to match duplicated words use the regex **(\w+)\s+\1 which matches a "word", then at least one whitespace, and then the same word as was captured.**
- In long or complicated regexes it is often more convenient to use names rather than numbers for captures. This can also make maintenance easier.
- To name a capture, follow the opening parenthes with **?P*<name>.***
  - ***Example: (?P<key>\w+)=(?P<value>.+) has two captures called*** "key" and "value".
- The syntax for backreferences to named captures inside a regex is **(?P=*name).***
  - ***Example: (?P<word>\w+)\s+(?P=word) matches duplicate*** words using a capture called "word".

Assertions and Flags

- One problem that affects many of the regexes is that they can match more or different text than we intended.

- For example, the regex **aircraft|airplane|jet will match "waterjet" and "jetski" as well as "jet".**
  - This kind of problem can be solved by using assertions.

- An assertion does not match any text, but instead says something about the text at the point where the assertion occurs.

  - One assertion is **\b (word boundary), which asserts that the character that precedes** it must be a "word" (**\w) and the character that follows it must be a non-**"word" (**\W), or vice versa.**

  - In the context of the original regex, write it either as **\baircraft\b|\bairplane\b|\bjet\b or \b(?:aircraft|airplane|jet)\b,** that is, word boundary, noncapturing expression, word boundary.

Regular Expression Assertions

Symbol	Meaning
^	Matches at the start; also matches after each newline with the `re.MULTILINE` flag
$	Matches at the end; also matches before each newline with the `re.MULTILINE` flag
\A	Matches at the start
\b	Matches at a "word" boundary; influenced by the `re.ASCII` flag—inside a character class this is the escape for the backspace character
\B	Matches at a non-"word" boundary; influenced by the `re.ASCII` flag
\Z	Matches at the end
(?=*e*)	Matches if the expression *e* matches at this assertion but does not advance over it—called *lookahead* or *positive lookahead*
(?!*e*)	Matches if the expression *e* does not match at this assertion and does not advance over it—called *negative lookahead*
(?<=*e*)	Matches if the expression *e* matches immediately before this assertion—called *positive lookbehind*
(?<!*e*)	Matches if the expression *e* does not match immediately before this assertion—called *negative lookbehind*

Assertions and Flags (contd..)

- the *key=value regex with comments:*

```
^[ \t]*                 # start of line and optional leading whitespace
(?P<key>\w+)            # the key text
[ \t]*=[ \t]*           # the equals with optional surrounding whitespace
(?P<value>[^\n]+)       # the value text
(?<![ \t])              # negative lookbehind to avoid trailing whitespace
```

- In the case of the *key=value regex, the negative lookbehind assertion means* that at the point it occurs the *preceding character must not be a space or a tab.*

  - **Example**: To read a multiline text that contains the names “Helen Patricia Sharman”, “Jim Sharman”, “Sharman Joshi”, “Helen Kelly”, and so on, and match “Helen Patricia”, but only when referring to “Helen Patricia Sharman”.
    - **Use \b(Helen\s+Patricia)\s+Sharman\b**
    - the same thing using a lookahead assertion, is**\b(Helen\s+Patricia)(?=\s+Sharman\b).**

Assertions and Flags (contd..)

- The final piece of regex syntax that Python's regular expression engine offers is a means of setting the flags.

- The flags are set by passing them as additional parameters when calling the re.compile() function, but sometimes it is more convenient to set them as part of the regex itself.

- The syntax is simply **(?*flags)* where** *flags is one or more of*
  - *a (the same as passing re.ASCII)*
  - i (re.IGNORECASE)
  - m (re.MULTILINE)
  - s (re.DOTALL), and
  - x (re.VERBOSE)

- If the flags are set this way they should be put at the start of the regex; they match nothing, so their effect on the regex is only to set the flags.

The Regular Expression Module

- The re module provides two ways of working with regexes:
  - One is to use the functions where each function is given a regex as its first argument.
  - Another way of writing this regex would be to use the character class **[\dA-F]** and pass the re.IGNORECASE flag as the last argument to the re.compile() call, or to use the regex **(?i)#[\dA-F]{6}\b which starts with the ignore case flag.**

The Regular Expression Module's Functions

Syntax	Description
`re.compile( r, f)`	Returns compiled regex `r` with its flags set to `f` if specified. (The flags are described in Table 13.5.)
`re.escape(s)`	Returns string `s` with all nonalphanumeric characters backslash-escaped—therefore, the returned string has no special regex characters
`re.findall( r, s, f)`	Returns all nonoverlapping matches of regex `r` in string `s` (influenced by the flags `f` if given). If the regex has captures, each match is returned as a tuple of captures.
`re.finditer( r, s, f)`	Returns a match object for each nonoverlapping match of regex `r` in string `s` (influenced by the flags `f` if given)
`re.match( r, s, f)`	Returns a match object if the regex `r` matches at the start of string `s` (influenced by the flags `f` if given); otherwise, returns `None`
`re.search( r, s, f)`	Returns a match object if the regex `r` matches anywhere in string `s` (influenced by the flags `f` if given); otherwise, returns `None`
`re.split( r, s, m, f)`	Returns the list of strings that results from splitting string `s` on every occurrence of regex `r` doing up to `m` splits (or as many as possible if no `m` is given, and for Python 3.1 influenced by flags `f` if given). If the regex has captures, these are included in the list between the parts they split.
`re.sub( r, x, s, m, f)`	Returns a copy of string `s` with every (or up to `m` if given, and for Python 3.1 influenced by flags `f` if given) match of regex `r` replaced with `x`—this can be a string or a function; see text
`re.subn( r, x, s m, f)`	The same as `re.sub()` except that it returns a 2-tuple of the resultant string and the number of substitutions that were made

The Regular Expression Module's Flags

Flag	Meaning
re.A or re.ASCII	Makes \b, \B, \s, \S, \w, and \W assume that strings are ASCII; the default is for these character class shorthands to depend on the Unicode specification
re.I or re.IGNORECASE	Makes the regex match case-insensitively
re.M or re.MULTILINE	Makes ^ match at the start and after each newline and $ match before each newline and at the end
re.S or re.DOTALL	Makes . match every character including newlines
re.X or re.VERBOSE	Allows whitespace and comments to be included

The re.compile() method

- Combine a regular expression pattern into pattern objects, which can be used for pattern matching.
- It also helps to search a pattern again without rewriting it.
- Example:

```
import re
pattern=re.compile('[A-Z][a-z]+')
result=pattern.findall('Learning is all about excellence')
print(result)

pattern=re.compile('TP')
result=pattern.findall('TP Tutorialspoint TP')
print(result)
result2=pattern.findall('TP is most popular tutorials site of India')
print(result2)
```

Output:
['Learning']
['TP', 'TP']
['TP']

Matching Versus Searching

- The difference is in the handling span of data between ‘re.match’ and ‘re.search’.
- ‘re.match’ scans only at the beginning of the string for matching the regular expressions. If it detect similarity in the pattern of the beginning of the string and regular expression, it executes.
- In contrast ‘re.search’ scans throughout the string for the match in the string and regular expressions.
- Example:

```
import re
a = "123abc"
t = re.match("[a-z]+",a)
y = re.search("[a-z]+",a)
print (t)
print (y)
```

Output:

```
None
<re.Match object; span=(3, 6), match='abc'>
```

Replace

- string.replace(s, old, new[, maxreplace])

- These are the function parameters:
  - **s**: The string required to be searched and replaced.
  - **old**: The old sub-string we want to replace.
  - **new**: The new sub-string we want to replace in place of old one.
  - **max replace**: The maximum number of times the sub-string is required to be replaced.

- Example:

```
our_str = 'Spider man'
new_str = our_str.replace('Spider', 'Bat')
print(new_str)
new_str = our_str.replace('man', 'Sense')
print(new_str)
```

Output:

Bat man
Spider Sense

Split

- The re.split function returns a list where the string has been split at each match.

- Example:

```
#part1
x = 'wind,water,fire'
k = x.split(",")
print (k)
#part2
a,b,c = x.split(",")
print (a,b,c)
```

Output:

```
['wind', 'water', 'fire']
wind water fire
```

sub

- The re.sub function replaces the matches with the text of your choice.

- Example:

```
import re
s = 'aaa@xxx.com bbb@yyy.com ccc@zzz.com'
print(re.sub('[a-z]*@', 'ISE@', s))
```

Output:

ISE@xxx.com ISE@yyy.com ISE@zzz.com

subn

- re.subn function is similar to re.sub function but it returns a tuple of items containing the new string and the number of substitutions made.

- Example:

```
import re
s = 'aaa@xxx.com bbb@yyy.com ccc@zzz.com'
print(re.subn('[a-z]*@', 'ISE@', s))
```

Output:

('ISE@xxx.com ISE@yyy.com ISE@zzz.com', 3)

escape

- re.escape returns *string* with all non-alphanumerics backslashed.

- this is useful to match an arbitrary literal string that may have regular expression metacharacters in it.

- Example:

```
import re
s = re.escape("Hello 123 .?!@ World")
print(s)
```

Output:

```
Hello\ 123\ \.\?!@\ World
```

findall

- The re.findall() helps to get a list of all matching patterns.
- It searches from start or end of the given string.
- Example:

```
import re
abc = 'ise@google.com, students@hotmail.com, users@yahoomail.com'
emails = re.findall(r'[\w\.-]+@[\w\.-]+', abc)
for email in emails:
    print(email)
```

Output:

ise@google.com
students@hotmail.com
users@yahoomail.com

finditer

- re.finditer(pattern, string, flags=0)
- It returns an iterator yielding MatchObject instances over all non-overlapping matches for the RE pattern in string.
- The string is scanned left-to-right, and matches are returned in the order found.
- Empty matches are included in the result.
- Example:

```
import re
s1 = 'Blue Berries'
pattern = 'Blue Berries'
for match in re.finditer(pattern, s1):
   s = match.start()
   e = match.end()
   print('String match "%s" at %d:%d' % (s1[s:e], s, e))
```

Output:

String match "Blue Berries" at 0:12

Regular Expression Object Methods

Syntax	Description
`rx.findall(s [, start, end])`	Returns all nonoverlapping matches of the regex in string `s` (or in the *start*:*end* slice of `s`). If the regex has captures, each match is returned as a tuple of captures.
`rx.finditer(s [, start, end])`	Returns a match object for each nonoverlapping match in string `s` (or in the *start*:*end* slice of `s`)
`rx.flags`	The flags that were set when the regex was compiled
`rx.groupindex`	A dictionary whose keys are capture group names and whose values are group numbers; empty if no names are used
`rx.match(s, [start, end])`	Returns a match object if the regex matches at the start of string `s` (or at the start of the *start*:*end* slice of `s`); otherwise, returns `None`
`rx.pattern`	The string from which the regex was compiled
`rx.search(s, [start, end])`	Returns a match object if the regex matches anywhere in string `s` (or in the *start*:*end* slice of `s`); otherwise, returns `None`
`rx.split(s, m)`	Returns the list of strings that results from splitting string `s` on every occurrence of the regex doing up to `m` splits (or as many as possible if no `m` is given). If the regex has captures, these are included in the list between the parts they split.
`rx.sub(x, s, m)`	Returns a copy of string `s` with every (or up to `m` if given) match replaced with `x`—this can be a string or a function; see text
`rx.subn(x, s m)`	The same as `re.sub()` except that it returns a 2-tuple of the resultant string and the number of substitutions that were made

Regular Expression Objects

- Python regular expression objects are compiled regular expressions created by re.compile.

- The five most commonly used regular expression methods are:
  - search
  - match
  - split
  - findall
  - Sub

- Each of these methods on a compiled regular expression object has a corresponding function which can be applied to strings.

Match Object Attributes and Methods

Syntax	Description
`m.end(g)`	Returns the end position of the match in the text for group *g* if given (or for group 0, the whole match); returns -1 if the group did not participate in the match
`m.endpos`	The search's end position (the end of the text or the *end* given to `match()` or `search()`)
`m.expand(s)`	Returns string `s` with capture markers (`\1`, `\2`, `\g<name>`, and similar) replaced by the corresponding captures
`m.group(g, ...)`	Returns the numbered or named capture group `g`; if more than one is given a tuple of corresponding capture groups is returned (the whole match is group 0)
`m.groupdict(default)`	Returns a dictionary of all the named capture groups with the names as keys and the captures as values; if a *default* is given this is the value used for capture groups that did not participate in the match
`m.groups(default)`	Returns a tuple of all the capture groups starting from 1; if a *default* is given this is the value used for capture groups that did not participate in the match

Match Object Attributes and Methods

`m.lastgroup`	The name of the highest numbered capturing group that matched or `None` if there isn't one or if no names are used
`m.lastindex`	The number of the highest capturing group that matched or `None` if there isn't one
`m.pos`	The start position to look from (the start of the text or the *start* given to `match()` or `search()`)
`m.re`	The regex object which produced this match object
`m.span(g)`	Returns the start and end positions of the match in the text for group *g* if given (or for group 0, the whole match); returns `(-1, -1)` if the group did not participate in the match
`m.start(g)`	Returns the start position of the match in the text for group *g* if given (or for group 0, the whole match); returns -1 if the group did not participate in the match
`m.string`	The string that was passed to `match()` or `search()`

Example 1

Write a python script to return the first word of the given string.

```
import re
result=re.findall(r‘^\w+','AV is largest Analytics”)
print (result)
```

```
Output:
['AV']
```

Example 2

Write a python script to return the first two character of each word in a given string.

```
import re
result=re.findall(r'\b\w.','AV is largest Analytics
community of India')
print (result)
```

```
Output:
['AV', 'is', 'la', 'An', 'co', 'of', 'In']
```

Thank you