

Python Assignment_1 - Mitushi Vishwakarma

Introduction to Python

Python is a high level, interpreted, dynamically typed, object oriented programming language.

VARIABLES : names that refer to values or objects in memory.

Variables

```
In [5]: x = 10
        name = "John"
        Bool = True
        a = 10.50
```

DATA TYPES : Variables can store data of different types. Python is dynamically typed, which means that the type of a variable is inferred at runtime.

Data Types

1. Numeric : int, float, complex

```
In [17]: # int: Integer data type represents whole numbers.
        age = 25
        print(type(age))

<class 'int'>
```

```
In [13]: # float: Floating-point data type represents decimal numbers.
        temp = 98.6
        print(type(temp))

<class 'float'>
```

```
In [14]: # complex : Complex numbers have a real and imaginary part.
        complex_num = 2 + 3j
        print(type(complex_num))

<class 'complex'>
```

2. Text : str

```
In [15]: # string : String data type represents sequences of characters.
name = "Alice"
print(type(name))

<class 'str'>
```

3. Boolean : bool

```
In [16]: # Boolean : Boolean data type represents truth values, either True or False.
Boolean = True
print(type(Boolean))

<class 'bool'>
```

4. Sequence : list, tuple

```
In [18]: # list : A list is an ordered collection that can contain elements of different data types.
my_list = [1, 2, "three", 4.0]
print(type(my_list))

<class 'list'>
```

```
In [19]: # tuple : A tuple is an ordered, immutable collection.
my_tuple = (1, 2, "three", 4.0)
print(type(my_tuple))

<class 'tuple'>
```

OPERATORS : Operators in Python are symbols or keywords that perform operations on operands. Operands can be variables, literals, or expressions.

1. Arithmetic operators

```
In [20]: a = 10
b = 3
print(a + b)
print(a - b)
print(a * b)
print(a / b)
print(a % b)
print(a ** b)
print(a // b)

13
7
30
3.3333333333333335
1
1000
3
```

2. Assignment operators

```
In [22]: x = 5  
y = 7  
x += 3  
y -= 3  
print(x,y)
```

8 4

3. Comparison operators

```
In [23]: x = 5  
y = 3  
print(x == y)  
print(x != y)  
print(x <= y)  
print(x >= y)
```

False

True

False

True

4. Logical Operators

```
In [24]: a = True  
b = False  
print(a and b)  
print(a or b)  
print(not a)
```

False

True

False

5. Bitwise Operators

```
In [25]: a = 10  
b = 4  
print(a & b)  
print(a | b)  
print(~a)  
print(a ^ b)  
print(a >> 2)  
print(a << 2)
```

```
0  
14  
-11  
14  
2  
40
```

CONTROL STRUCTURE : Control structures in programming determine the flow of execution based on conditions. if, else, elif are examples of control structures.

Control Structure

```
In [29]: # if

i = 10

if (i > 15):
    print("10 is less than 15")

print("Outside if")
```

Outside if

```
In [30]: # if-else

i = 20
if (i < 15):
    print("if Block")
    print(f"{i} is smaller than 15")
else:
    print("else Block")
    print(f"{i} is greater than 15")

print("not in if and not in else Block")
```

else Block
20 is greater than 15
not in if and not in else Block

```
In [31]: # if-elif-else

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

i is 20

FOR LOOP : The for loop is used for iterating over a sequence (like a list, tuple, or string) or other iterable objects.

```
In [32]: # For loop
for letter in "Hello World":
    print(letter)
```

```
H
e
l
l
o

W
o
r
l
d
```

WHILE LOOP : The while loop continues to execute a block of code as long as a specified condition is true.

```
In [1]: # While Loop
i = 1
while i < 6:
    print(i)
    i += 1
```

```
1
2
3
4
5
```

NESTED LOOP : A nested loop is a loop inside another loop. It allows for more complex iteration patterns.

```
In [2]: # Nested Loop
lst=[]
for x in [1,2,3]:
    for y in [10,100,1000]:
        lst.append(x*y)
lst
```

```
Out[2]: [10, 100, 1000, 20, 200, 2000, 30, 300, 3000]
```

Break, Continue & Pass:

- break: Terminates the loop.

```
In [3]: # break
        fruits = ["apple", "banana", "cherry"]
        for x in fruits:
            print(x)
            if x == "banana":
                break
```

```
apple
banana
```

- continue: Skips the rest of the loop and goes to the next iteration.

```
In [5]: # continue
        fruits = ["apple", "banana", "cherry"]
        for x in fruits:
            if x == "banana":
                continue
            print(x)
```

```
apple
cherry
```

- pass: Used as a placeholder where syntactically some code is required but no action is desired.

```
In [4]: #pass
        for x in [0, 1, 2]:
            pass
```

INPUT AND OUTPUT : In Python, you can use the input() function to receive input from the user, and the print() function to display output to the console.

```
In [6]: user_input = input("Enter something: ")
        print("You entered:", user_input)
```

```
Enter something: 3
You entered: 3
```

LISTS, Slicing and Methods : Lists are used to store multiple items in a single variable. A list is a built-in data type that represents an ordered and mutable collection of elements. Lists are defined using square brackets [] and can be modified after creation.

```
In [7]: my_list=[1,2,3]
```

```
In [8]: my_list=[1,'string',23.4]
```

Accessing Elements: using negative or positive indexing

```
In [10]: first_element = my_list[0] # Accessing the first element  
second_element = my_list[1] # Accessing the second element  
print(first_element,second_element)
```

1 string

```
In [11]: last_element = my_list[-1]  
print(last_element)
```

23.4

Modifying Lists:

Changing an Element:

```
In [13]: my_list=[1,'string',23.4]  
my_list[1] = 10  
print(my_list)
```

[1, 10, 23.4]

Adding Elements:

- append()

```
In [14]: my_list.append(6)  
print(my_list)
```

[1, 10, 23.4, 6]

- Insert() :

```
In [15]: my_list.insert(2, 7)
         print(my_list)

[1, 10, 7, 23.4, 6]
```

Removing Elements:

- Remove():

```
In [17]: my_list.remove(6)
         print(my_list)

[1, 10, 7, 23.4]
```

- pop()

```
In [18]: popped_element = my_list.pop(1)
         print(my_list)

[1, 7, 23.4]
```

Sorting Lists:

```
In [19]: num_list = [1,9,334,90,2]
         num_list.sort()
         num_list
```

```
Out[19]: [1, 2, 9, 90, 334]
```

Reverse () :

```
In [20]: num_list.reverse()
         num_list
```

```
Out[20]: [334, 90, 9, 2, 1]
```

Slicing : You can extract a sublist (slice) from a list using slicing notation. Slicing is a powerful feature in Python that allows you to extract a portion of a list. The syntax for slicing is `list[start:stop:step]`.

```
In [21]: subset = my_list[1:4]
subset
```

```
Out[21]: [7, 23.4]
```

```
In [22]: num_list[::-1]
```

```
Out[22]: [1, 2, 9, 90, 334]
```

DICTIONARIES and Methods : A dictionary is a built-in data type that represents an unordered collection of key-value pairs. Each key in a dictionary must be unique, and it is associated with a specific value. Dictionaries are defined using curly braces {}, and key-value pairs are separated by colons ‘:’.

```
In [23]: my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
my_dict
```

```
Out[23]: {'name': 'John', 'age': 25, 'city': 'New York'}
```

Accessing Values:

```
In [25]: name_value = my_dict['name']
age_value = my_dict['age']
print(name_value, age_value)
```

```
John 25
```

Modifying and Adding Key-Value Pairs:

Dictionaries are mutable, so you can modify existing values or add new key-value pairs.

```
In [26]: my_dict['age'] = 26 # Modifying the value associated with the 'age' key
my_dict['gender'] = 'Male' # Adding a new key-value pair
my_dict
```

```
Out[26]: {'name': 'John', 'age': 26, 'city': 'New York', 'gender': 'Male'}
```

Removing Key-Value Pairs:

`pop(key)`: Removes the key and its associated value from the dictionary and returns the value.

```
In [27]: removed_age = my_dict.pop('age')
my_dict
```

```
Out[27]: {'name': 'John', 'city': 'New York', 'gender': 'Male'}
```

`popitem()`: Removes and returns the last key-value pair as a tuple.

```
In [28]: last_item = my_dict.popitem()
my_dict
```

```
Out[28]: {'name': 'John', 'city': 'New York'}
```

`keys()`: Returns a view of all the keys in the dictionary.

```
In [30]: keys_list = my_dict.keys()
keys_list
```

```
Out[30]: dict_keys(['name', 'city'])
```

`values()`: Returns a view of all the values in the dictionary.

```
In [31]: values_list = my_dict.values()
values_list
```

```
Out[31]: dict_values(['John', 'New York'])
```

`items()`: Returns a view of all key-value pairs as tuples.

```
In [33]: items_list = my_dict.items()
print(items_list)

dict_items([('name', 'John'), ('city', 'New York')])
```

get(key, default): Returns the value associated with the specified key. If the key is not found, it returns the default value.

```
In [34]: age_value = my_dict.get('age', 0) # Returns the age value or 0 if 'age' is not in the dictionary
age_value
Out[34]: 0
```

update(dictionary): Updates the dictionary with key-value pairs from another dictionary.

```
In [35]: other_dict = {'gender': 'Male', 'city': 'San Francisco'}
my_dict.update(other_dict)
my_dict
Out[35]: {'name': 'John', 'city': 'San Francisco', 'gender': 'Male'}
```

Introduction to Set & Set Methods :

In Python, a set is an unordered collection of unique elements. Sets are defined using curly braces {}, and elements are separated by commas. Sets are useful when you want to work with a collection of distinct items and perform operations like union, intersection, and difference.

```
In [44]: # Creating a set
my_set = {1, 2, 3, 4, 5}
my_set
```

```
Out[44]: {1, 2, 3, 4, 5}
```

```
In [46]: # Adding elements to a set
my_set.add(6)
my_set.update([7, 8, 9])
my_set
```

```
Out[46]: {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
In [47]: # Removing elements from a set
my_set.remove(3)
my_set.discard(10) # No error if the element is not present
my_set
```

```
Out[47]: {1, 2, 4, 5, 6, 7, 8, 9}
```

```
In [49]: # Set operations
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
union_set
```

```
Out[49]: {1, 2, 3, 4, 5}
```

```
In [50]: intersection_set = set1.intersection(set2)
intersection_set
```

```
Out[50]: {3}
```

```
In [51]: difference_set = set1.difference(set2)
difference_set
```

```
Out[51]: {1, 2}
```

Introduction to Map & Map Methods

Mapping function : A mapping function is a function that transforms each element of an iterable using a specified rule. The `map()` function in Python applies a given function to all items in an input list (or any other iterable) and returns an iterator that produces the results.

```
In [39]: def square(x):
          return x ** 2

          numbers = [1, 2, 3, 4]
          squared_numbers = map(square, numbers)
          result_list = list(squared_numbers)
          print(result_list)

          [1, 4, 9, 16]
```

String Function : There are various built-in string functions in Python for string manipulation, such as len(), lower(), upper(), strip(), split(), join(), etc.

```
In [54]: my_string = "Hello, World!"
          print(len(my_string))
          print(my_string.lower())
          print(my_string.split(','))
          new_string = "python programming"
          capitalized_string = new_string.capitalize()
          print(capitalized_string)
          titlecased_string = new_string.title()
          print(titlecased_string)

          13
          hello, world!
          ['Hello', ' World!']
          Python programming
          Python Programming
```

Number Function : Mathematical operations and functions are available for numerical operations. Common functions include abs(), round(), max(), min(), sum(), etc.

```
In [42]: number = -5
          absolute_value = abs(number)
          rounded_value = round(3.14159)
          print(absolute_value, rounded_value)

          5 3
```

Date and Time Function

The datetime module in Python provides functions for working with dates and times. Common functions include datetime.now(), strftime(), strptime(), timedelta(), etc.

```
In [71]: from datetime import datetime, timedelta

current_time = datetime.now()
print(current_time)
formatted_time = current_time.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_time)
next_day = current_time + timedelta(days=1)
print(next_day)

2024-01-30 10:16:05.603412
2024-01-30 10:16:05
2024-01-31 10:16:05.603412
```

Python Functions

In Python, a function is a block of reusable code designed to perform a specific task. Functions are defined using the `def` keyword.

```
In [72]: def hello(name, age):
        print(f"Hello {name}, your age is {age}")
        return f"Hello {name}"

x=hello("nikita", 6)
print(x)

Hello nikita, your age is 6
Hello nikita
```

Default Argument Values

We can provide default values for function parameters. If a value is not provided when calling the function, the default value is used.

```
In [74]: def power(base, exponent=2):
        return base ** exponent

result1 = power(3)
print(result1)
result2 = power(3, 3)
print(result2)

9
27
```

Keyword Arguments

We can pass arguments to a function by explicitly specifying the parameter names

```
In [75]: def print_info(name, age):  
         print(f"Name: {name}, Age: {age}")  
  
         print_info(age=25, name="John")
```

Name: John, Age: 25

Special parameters

Python functions can have special parameters like `*args` and `**kwargs` to handle variable numbers of arguments.

```
In [76]: def print_args(*args, **kwargs):  
         print("Positional Arguments:", args)  
         print("Keyword Arguments:", kwargs)  
  
         print_args(1, 2, 3, name="John", age=25)
```

Positional Arguments: (1, 2, 3)

Keyword Arguments: {'name': 'John', 'age': 25}

Arbitrary Argument Lists

We can use `*args` to allow a function to accept any number of positional arguments.

```
In [77]: def sum_numbers(*args):  
         return sum(args)  
  
         result = sum_numbers(1, 2, 3, 4)  
         print(result)
```

10

Lambda Expressions

Lambda expressions, or anonymous functions, allow you to create small, one-time-use functions.

```
In [79]: multiply = lambda x, y: x * y  
         result = multiply(3, 4)  
         print(result)
```

12

OOPS

Class and Object : In Python, a class is a blueprint for creating objects, and an object is an instance of a class. Classes define attributes (characteristics) and methods (behaviors) that objects can have.


```
In [36]: # Pet class
class Pet:
    def __init__(self, name, age, breed):
        self.Name = name
        self.Age = age
        self.Breed = breed
p = Pet("Aayu",20,"Girl")
print(p)

<__main__.Pet object at 0x000001F020E31C30>
```

Access Specifiers : In Python, access specifiers like public, private, and protected are not explicitly defined, but naming conventions are used. Attributes and methods starting with a single underscore (_) are considered protected, and those starting with a double underscore (__) are considered private.

```
In [37]: class MyClass:
    def __init__(self):
        self.public_attribute = 10
        self._protected_attribute = 20
        self.__private_attribute = 30
```

Constructor : A constructor is a special method in a class that is automatically called when an object is created. In Python, the constructor is named `__init__`.

```
In [36]: # Pet class
class Pet:
    def __init__(self, name, age, breed):
        self.Name = name
        self.Age = age
        self.Breed = breed
```

Inheritance : Inheritance allows a new class (subclass/derived class) to inherit attributes and methods from an existing class (base class/parent class).

```
In [57]: # Inheritance
class Parent:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # self is given because whenever any object calls any function self is passed as an argument automatically
    def color_of_eye(self):
        print("Black")

class Child(Parent):
    def color_of_eye(self):
        print("brown")

In [58]: c1 = Child("mitu" , 22)
c1.color_of_eye()

brown
```

Polymorphism : Polymorphism allows objects of different types to be treated as objects of a common type. In Python, polymorphism is achieved through method overloading or operator overloading.

```
In [59]: # Polymorphism

class Animal:
    def bark(self):
        print("Animals bark")
class Dog(Animal):
    def bark(self):
        print("Dogs bark")
class Fish(Animal):
    def bark(self):
        print("Fish doesn't bark")

# def can_bark(obj):
#     obj.bark()

animals = [Dog(), Fish()]
for animal in animals:
    animal.bark()

Dogs bark
Fish doesn't bark
```

Method Overriding : Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

```
In [60]: # Method Overriding
class Animal:
    def speak(self):
        return "Animal speaks"

class Dog(Animal):
    def speak(self):
        return "Dog barks"
```

File handling : File handling in Python involves operations like opening, reading, writing, and closing files. The open() function is commonly used.

```
In [62]: myfile = open('myfile.txt')
type(myfile)
```

```
Out[62]: _io.TextIOWrapper
```

```
In [63]: myfile.read()
```

```
Out[63]: 'Hello !!\nFirst line\nSecond line\nThird Line'
```

```
In [64]: myfile.read() # cursor moved to last in the above code
```

```
Out[64]: ''
```

```
In [65]: myfile.seek(0) # moving cursor back to start
myfile.read()
```

```
Out[65]: 'Hello !!\nFirst line\nSecond line\nThird Line'
```

```
In [67]: myfile.seek(0)
myfile.readlines()
```

```
Out[67]: ['Hello !!\n', 'First line\n', 'Second line\n', 'Third Line']
```

```
In [68]: myfile.close()
```

```
In [69]: with open('myfile.txt') as newfile:
    contents = newfile.read()
    print(contents)
```

```
Hello !!
First line
Second line
Third Line
```

EXCEPTION HANDLING : Exception handling allows us to handle errors gracefully. It involves try, except, finally, and optional else blocks.

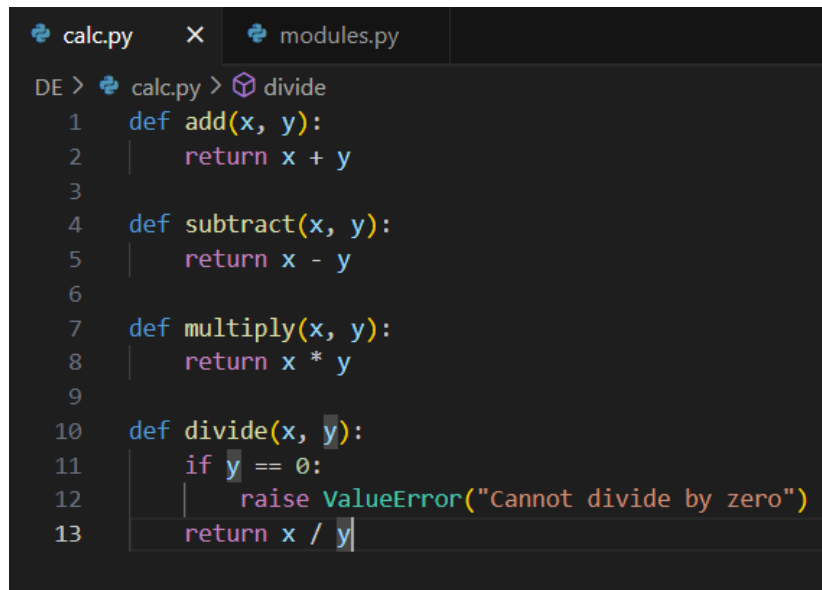
```
In [3]: try :
        num = int(input("Enter a number :"))
        a = 100/num
        print("Hi")
        print(a)
    except ZeroDivisionError:
        print("Zero can't divide any number")
    except ValueError:
        print("Some error occurred")
    else:
        print("no exception")
    finally:
        print("executed always")
```

```
Enter a number :3
Hi
33.333333333333336
no exception
executed always
```

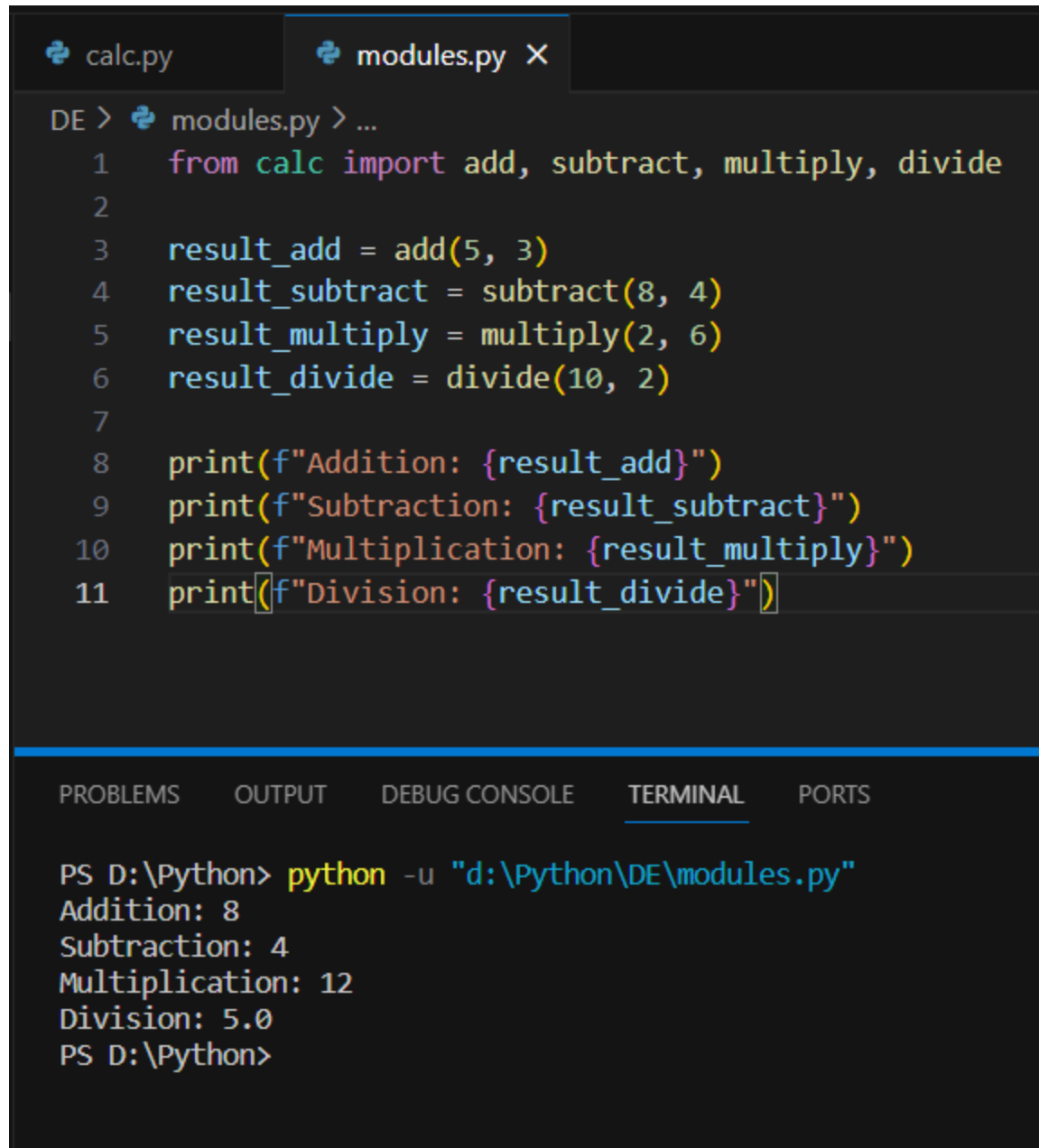
Python Modules:

A module in Python is a file containing Python definitions and statements. The file name is the module name with the suffix ".py" added. Modules allow you to organize your Python code logically and make it more reusable.

Example : A module for calculator with functions for different operations.



```
calc.py  X  modules.py
DE > calc.py > divide
1  def add(x, y):
2      return x + y
3
4  def subtract(x, y):
5      return x - y
6
7  def multiply(x, y):
8      return x * y
9
10 def divide(x, y):
11     if y == 0:
12         raise ValueError("Cannot divide by zero")
13     return x / y
```



The image shows a Python IDE interface. At the top, there are two tabs: 'calc.py' and 'modules.py'. The 'modules.py' tab is active, showing a Python script. The script imports functions from a module named 'calc' and performs four calculations: addition, subtraction, multiplication, and division. The results are printed using f-strings. Below the code editor, there is a terminal window with tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL', and 'PORTS'. The 'TERMINAL' tab is selected, showing the command to run the script and its output.

```
DE > modules.py > ...
1  from calc import add, subtract, multiply, divide
2
3  result_add = add(5, 3)
4  result_subtract = subtract(8, 4)
5  result_multiply = multiply(2, 6)
6  result_divide = divide(10, 2)
7
8  print(f"Addition: {result_add}")
9  print(f"Subtraction: {result_subtract}")
10 print(f"Multiplication: {result_multiply}")
11 print(f"Division: {result_divide}")
```

```
PS D:\Python> python -u "d:\Python\DE\modules.py"
Addition: 8
Subtraction: 4
Multiplication: 12
Division: 5.0
PS D:\Python>
```

Standard Modules:

Python comes with a set of standard modules that provide various functionalities.

```
1  from math import sqrt,factorial
2
3  print(factorial(5))
4  print(sqrt(144))
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

```
PS D:\Python> python -u "d:\Python\DE\StandardModules.py"
120
12.0
PS D:\Python>
```