CREATING AND USING MODULES — Progress In Python 4.1

This Prip session is based on practice for creating and using modules.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PrIP 4.1 under Chapter 4 after practically doing it on the computer.

>>>❖<<<

4.4 USING PYTHON STANDARD LIBRARY'S FUNCTIONS AND MODULES

Python's standard library is very extensive that offers many built-in functions that you can use without having to import any library. Python's standard library is by default available, so you don't need to import it separately.

Python's standard library also offers, other that the built-in functions, some modules for specialized type of functionality, such as ***math*** module for mathematical functions ; ***random*** module for pseudo-random number generation ; ***urllib*** for functionality for adding and using web sites' address from within program ; etc.

In this section, we shall discuss how you can use Python's built-in functions and import and use various modules and Python's standard library.

4.4.1 Using Python' Built-in Functions

The Python interpreter has a number of functions built into it that are always available ; you need not import any module for them. In other words, the built in functions are part of current namespace of Python interpreter. So you use built-in functions of Python directly as :

```
<function-name>()
```

For example, the functions that you have worked with uptill now such as *input( ), int( ), float( ), type( ), len( )* etc. are all built in functions, that is why you never prefixed them with any module name.

4.4.1A Python's built-in Mathematical Functions

Python provides many mathematical built-in functions. You have worked with many built-in mathematical functions of Python in your previous class. Let us quickly recall those and then we shall talk about some more such functions.

Function name	Description
`len( )`	Returns the length of a sequence or iterable *e.g.*, len("abc") gives 3.
`pow( )`	Returns a^b when a and b are given as arguments, *e.g.*, pow(3, 4) gives 81.
`str( )`	Converts a number to a string, *e.g.*, str(12) will give '12' and str(12.4) will give '12.4'.
`int( )`	Converts an integer-convertible string to integer, *e.g.*, int('12') will give 12.
`float( )`	Converts a float-convertible string to integer, *e.g.*, float('12.2') will give 12.2.
`range( )`	Returns an immutable sequence type, *e.g.*, range(3) will give sequence 0, 1, 2.
`type( )`	Returns the data type of passed argument, *e.g.*, type(12) will give <class 'int'>.

Scanned with CamScanner

Let us now talk about some more built-in mathematical functions.

Table 4.1 Some useful built-in mathematical functions

Function name	Description	Examples

abs(x)	Takes an integer or a floating point number as argument and returns the absolute value of a number.	>>> abs(-12)
		12
		>>> abs(-12.4)
		12.4
		>>> abs(12.4)
		12.4

divmod(a, b)	Takes two (non-complex) numbers as arguments and returns a pair of numbers consisting of their quotient and remainder.
	For integers, the result is the same as (*a* // *b*, *a* % *b*). For floating point numbers the result is (*q*, *a* % *b*)
		>>> divmod(7, 2)
		(3, 1) *Pair of quotient and remainder returned*
		>>> divmod(7.25, 2.5)
		(2.0, 2.25)
		>>>

m(iterable)
m(iterable,arg)
	Returns sum of the items of an ***iterable*** from left to right and returns the total.
	With two arguments iterable and arg, it returns the sum of the items of an ***iterable*** and the ***arg's*** value.
	The ***iterable's*** items are normally numbers, and the ***arg*** value should be a number.
	(Recall that all sequence types are iterable.)
		>>> sum([2, 3, 4])
		9 *Sum of the elements of the iterable returned*
		>>> sum((2.5, 6, 4.2))
		12.7
		>>> sum([2, 3, 4], 5)
		14 *Sum of the elements of the iterable along with the argument value returned*
		>>> sum([2, 3, 4], 8.3)
		17.3

iterable)
arg1, arg2,….)
	Returns the largest item in an iterable / sequence or the largest of two or more arguments.
	If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned.
		>>> max(3, 5)
		5 *Maximum of two integer values returned*
		>>> max([3, 5, 9,], [7])
		[7] *Maximum of two list arguments returned – list that begins with a higher value is returned*
		>>> max((3, 5, 9, 10), (7,))
		(7,) *Maximum of two tuple arguments returned – tuple that begins with a higher value is returned*
		>>> max((3, 5, 9, 10))
		10 *Maximum of one iterable argument returned highest element from the tuple iterable*
		Maximum of one iterable argument >>> max([13, 15, 27, 10])

Scanned with CamScanner

Function name	Description	Examples

`min(iterable)`
`min(arg1, arg2,...)`
	Returns the smallest item in an iterable / sequence or the smallest of two or more arguments.
	If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned.
		`>>> min(3, 5)`
		`3` — *Minimum of two integer values returned*
		`>>> min([3, 5, 9, ], [7])`
		`[3, 5, 9]` — *Minimum of two list arguments returned – list that begins with a higher value is returned*
		`>>> min( (3, 5, 9, 10 ), (7,))`
		`(3, 5, 9, 10)` — *Minimum of two tuple arguments returned – tuple that begins with a higher value is returned*
		`>>> min( [13, 15, 27, 10 ])`
		`10` — *Minimum of one iterable argument returned – smallest element from the tuple iterable*
		`>>> min([13, 15, 27, 10])`
		`10` — *Minimum of one iterable argument returned – smallest element from the list iterable*

`oct(<integer>)`
`hex(<integer>)`
	returns octal string for given numbers *i.e.*, 0O + octal equivalent of number.
	returns hex string for given numbers *i.e.*, 0x + hexadecimal equivalent of number.
	Please note that *oct( ), hex( )* and *bin( )* do not return a number ; they return a string representation of converted number.
		`>>> n = 24`
		`>>> oct(n)`
		`'0o30'`
		`>>> hex(n)`
		`'0x18'`

Consider following program that uses two more built-in functions :

- **int (<number>)**[5] — truncates the fractional part of given number and returns only the integer or whole part.
- **round(<number>, [<ndigits>])** — returns number rounded to *ndigits* after the decimal points. If *ndigits* is not given, it returns nearest integer to its input.

4.1 Write a program that inputs a real number and converts it to nearest integer using two different built-in functions. It also displays the given number rounded off to 3 places after decimal.

rogram

```
num = float(input("Enter a real number:"))
tnum = int(num)
rnum = round(num)
print("Number", num, "converted to integer in 2 ways as", tnum, "and", rnum )
rnum2 = round(num, 3)
print(num, "rounded off to 3 places after decimal is", rnum2)
```

The int( ) can also convert a number string into an equivalent number *e.g.*, "1235" can be converted to number 1235 using int("1235") ; works with integer strings only.

Scanned with CamScanner

Sample run of above program is :

```
Enter a real number: 5.555682
Number 5.555682 converted to integer in 2 ways as 5 and 6
5.555682 rounded off to 3 places after decimal is 5.556
```

***The behaviour of round( ) can be surprising for floats, e.g., round(0.5) and round(–0.5) are 0, and round(1.5) is 2.*

4.4.1B Python's built-in String Functions

Let us now use some string functions. Although you have worked with many string functions in your previous class, let us use three new string based functions. These are :

- **<Str>.join(<string iterable>)** – joins a string or character (*i.e.*, <str>) after each member of the *string iterator i.e., a string based sequence.*
- **<Str>.split(<string /char>)** – splits a string (*i.e.*, <str>) based on given string or character (*i.e.*, <string/char>) and returns a list containing split strings as members.
- **<Str>.replace(<word to be replaced>, <replace word>)** – replaces a word or part of the string with another in the given string <str>.

Let us understand and use these string functions practically. Carefully go through the examples of these as given below :

<str>.join( )

(*i*) If the string **based** iterator is a string then the <str> is inserted after every character of the string, *e.g.,*

```
>>>"*".join("Hello")
'H*e*l*l*o'
```

See, a character is joined with each member of the given string to form the new string

```
>>>"***".join("TRIAL")
'T***R***I***A***L'
```

*See, a string("***" here) is joined with each member of the given string to form the new string*

(*ii*) If the string based iterator is a *list* or *tuple* of strings then, the given string/character is joined with each member of the list or tuple, BUT the tuple or list must have all membe as strings otherwise Python will raise an error.

```
>>>"$$".join(["trial", "hello"])
Out[7]: 'trial$$hello'
```

Given string ("$$") joined between the individual items of a string based list

```
>>>"###".join(("trial", "hello", "new"))
Out[8]: 'trial###hello###new'
```

Given string ("$$") joined between the individual items of a string based tuple

```
>>>"###".join((123, "hello", "new"))
Traceback (most recent call last):

  File "<ipython-input-11-a0be3b94faec>", line 1, in <module>
    "###".join((123, "hello", "new"))

TypeError: sequence item 0: expected str instance, int found
```

The sequence must contain all strings else Python will raise an error.

Scanned with CamScanner

<str>.split( )

(i) If you do not provide any argument to split then by default it will split the given string considering whitespace as a separator, *e.g.*,

```
>>>"I Love Python".split()
['I', 'Love', 'Python']

>>>"I Love Python".split(" ")
['I', 'Love', 'Python']
```

With or without whitespace, the output is just the same i.e., the list containing individual words

(ii) If you provide a string or a character as an argument to split( ), then the given string is divided into parts considering the given string/character as separator and separator character is not included in the split strings *e.g.*,

```
>>>"I Love Python".split("o")
['I L', 've Pyth', 'n']
```

The given string is divided from positions containing "o"

<str>.replace( )

```
In[ ]:"I Love Python".replace("Python", "Programming")
'I Love Programming'
```

Word 'Python' has been replaced with 'Programming' in given string

Program 4.2 Write a program that inputs a main string and then creates an encrypted string by embedding a short symbol based string after each character. The program should also be able to produce the decrypted string from encrypted string.

```
def encrypt(sttr, enkey):
    return enkey.join(sttr)
def decrypt(sttr, enkey):
    return sttr.split(enkey)
#-main-
mainString = input("Enter main string :")
encryptStr = input("Enter encryption key :")
enStr = encrypt(mainString, encryptStr)
deLst = decrypt(enStr, encryptStr)
# deLst is in the form of a list, converting it to string below
deStr="".join(deLst)
print("The encrypted string is", enStr)
print("String after decryption is :", deStr)
```

The sample run of the above program is as shown below :

```
Enter main string : My main string
Enter encryption key : @$
The encrypted string is M@$y@$ @$m@$a@$i@$n@$ @$s@$t@$r@$i@$n@$g
String after decryption is :  My main string
```

Scanned with CamScanner

4.4.2 Working with Some Standard Library Modules

Other than built-in functions, standard library also provides some modules having functionality for specialized actions. Let us learn to use some such modules. In the following lines we shall talk about how to use some useful functions of *random* and *string* modules[6] of Python's standard library.

4.4.2A Using Random Module

Python has a module namely **random** that provides random-number[7] generators. A random number in simple words means *a number generated by chance, i.e.*, randomly.

To use random number generators in your Python program, you first need to import module ***random*** using any import command, *e.g.*,

```
import random
```

Some most common random number generator functions in ***random module*** are :

random( )	it returns a random floating point number N in the range [0.0, 1.0), *i.e.*, $0.0 \le N < 1.0$. Notice that the number generated with *random( )* will always be less than 1.0. (only lower range-limit is inclusive). **Remember, it generates a floating point number.**
randint(a, b)	it returns a random integer N in the range (a, b), *i.e.*, $a \le N \le b$ (both range-limits are inclusive). Remember, it generates an integer.
random.uniform(a, b)	it returns a random floating point number N such that a <= N <= b for a <= b and b <= N <= a for b < a.
random.randrange(stop) random.randrange(start, stop[, step])	it returns a randomly selected element from range(start, stop, step).

Let us consider some examples. In the following lines we are giving some sample codes alo with their output.

1. To generate a random floating-point number between 0.0 to 1.0, simply use **random(**

```
>>> import random
>>> print(random.random())
0.022353193431
```

The output generated is between range [0.0, 1.0)

2. To generate a random floating-point number between range ***lower to upper*** using **rando**

   (*a*) multiply **random( )** with difference of upper limit with lower limit, *i.e.*, (up lower)

   (*b*) add to it lower limit

note that learning how to use modules is important for learning how to use libraries.

Scanned with CamScanner