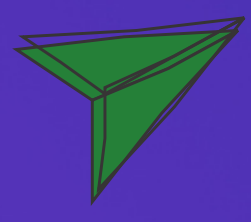
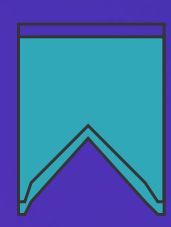
FUNCTION IN PYTHON

CREATED BY

UDAYABHANU NAYAK





By Udayabhanu Nayak CSE Grad 2024

Functions in one shot

Function

A function in programming is a reusable block of code that performs a specific task, typically taking input parameters and returning a result

- A function is a `block of organized code` written to carry out a specified task.
- Functions help break our program into `smaller` and `modular` chunks for better `readability`.
- Information can be passed into a function as arguments.
- Parameters are specified after the function name inside the parentheses.
- We can add as many parameters as we want. Parameters must be separated with a `comma`.
- A function `may or may not return data`.
- In Python a function is defined using the `def keyword`

Function has two parts

- Defination
- Call

In [1]:
```
### function defination , arguments

def function_name(argument1, argument2):
    '''
    This portion called DocString, acts as a comment that will not be
    '''
    # Actual code to be executed goes here
    print(argument1, argument2)

# Calling the function by name
obj = function_name('first', 'second')
```

```
first second
```

In Python,

you can define functions with default argument values.

Default arguments are used when the function is called without providing a value for a specific parameter.

In [2]:
```
def greet(name, greeting='Hello'):
    print(greeting, name)

# Calling the function with different arguments
greet('Udaya')
greet('Did you Follow me !', 'Hi')
```

```
Hello Udaya
Hi Did you Follow me !
```

4 ways of creating Function

NANR: No Arguments, No Return

This type of function doesn't take any arguments and doesn't return any value.

In [3]:
```
def greet():
    print("Knowledge is free quickly follow me 😍!")
    print("Without any argument , and Not using return ")

greet()
```

```
Knowledge is free quickly follow me 😍!
Without any argument , and Not using return
```

WAWR: With Arguments, With Return

This type of function takes arguments and returns a value.

In [4]:
```
def greet(name):
    message = "Knowledge is free quickly follow me, " + name + "!"
    print("Without argument , and using return ")
    return message


result = greet("Udaya 😁!")
print(result)
```

```
Without argument , and using return
Knowledge is free quickly follow me, Udaya 😁!!
```

WANR: With Arguments, No Return

This type of function takes arguments but doesn't return any value.

In [5]:
```
def greet(name):
    print("Knowledge is free quickly follow me 😊, " + name + "!")
    print("Using an argument, but not using return")


greet("Udaya !")
```

```
Knowledge is free quickly follow me 😊, Udaya !!
Using an argument, but not using return
```

NAWR: No Arguments, With Return

This type of function doesn't take any arguments but returns a value.

In [6]:
```
def greet():
    message = "Knowledge is free quickly follow me!  Udaya 😊"
    print("Without any argument , and using return ")
    return message

result = greet()
print(result)
```

```
Without any argument , and using return
Knowledge is free quickly follow me!  Udaya 😊
```

Types of Functions

- Built-in function :- Python predefined functions that are readily available for use like min() , max() , sum() , print() etc.
- User-Defined Functions:- Function that we define ourselves to perform a specific task.
- Anonymous functions : Function that is defined without a name. Anonymous functions are also called as lambda functions. They are not declared with the def keyword.

Anonymous function / Lambda Function

Lambda function can take any number of argument but can perform only one expression

It has no name

In [7]:
```
x = lambda a:a*2
print(x(10))
```

```
20
```

In [8]:
```
x = lambda a,b:a*b
print(x(10,20))
```

```
200
```

In [9]:
```
x = lambda a,b,c:a*b
print(x(10,20,30))
```

```
200
```

In [10]:
```
x = lambda a,b,c:a*b+c
print(x(10,20,30))
```

```
230
```

In [11]:
```
x = lambda a,b,c=2 : a+b+c
print(x(10,20,30))
```

```
60
```

Filter Function

Filter function returns an iterator where the items are filtered through a function to test if the item is acceptable or not

In [12]:
```
age = [23, 12, 23, 19, 18, 22, 11]

def check(x):
    if x > 18:
        return True
    else:
        return False

# Using filter to filter elements from age list based on the condition
obj = filter(check, age)

# Printing the filter object
print("It will print filter object:", obj)

# See items by using a loop

for i in obj:
    print(i,end=" ")
```

```
It will print filter object: <filter object at 0x0000023CCB19F850>
23 23 19 22
```

In [13]:
```
# Another way to see the items we need to convert it to a list

obj1=filter(check,age)

print("To see the items we need to convert it to a list:")
print(list(obj1))
```

```
To see the items we need to convert it to a list:
[23, 23, 19, 22]
```

Filter using lambda

In [14]:
```
age = [23, 12, 23, 19, 18, 22, 11]

obj = list(filter(lambda x : x>18 , age))
obj
```

Out[14]:
```
[23, 23, 19, 22]
```

Map function

Map function execute a specific function for each item in a iterable

In [15]:
```
no = [2,3,4,5,6]
def square(x):
    return x**2

obj = map(square , no)

# Printing the filter object
print("It will print Mapped object:", obj)

# See items by using a loop

for i in obj:
    print(i,end=" ")
```

```
It will print Mapped object: <map object at 0x0000023CCB2181F0>
4 9 16 25 36
```

In [16]:
```
# Another way to see the items we need to convert it to a list

obj1=map(square,no)

print("To see the items we need to convert it to a list:")
print(list(obj1))
```

```
To see the items we need to convert it to a list:
[4, 9, 16, 25, 36]
```

Map using lambda

In [17]:
```
no = [2,3,4,5,6]

obj = list(map(lambda x:x**2 , no))
print(obj)
```

```
[4, 9, 16, 25, 36]
```

Lambda Filter and Map in one

In [18]:
```
age = [20,90,36,11,75,33,27,30,24]

obj = list(filter(lambda x:x>29 , age))

obj2 = list(map(lambda x:x**2,obj))

print(obj2)
```

```
[8100, 1296, 5625, 1089, 900]
```

Reduce Function

Here first two element in a sequence is picked and apply to function in a commutative way .

Reduce is defined in functool module

In [19]:
```
from functools import reduce

n = [1,2,3]

def mul(x,y):
    return x*y

pro = reduce(mul,n)
print(pro)
```

```
6
```

Scope of a function

Scope is a region of program where a perticular identifier is accessible

Types

Local Scope / Functional Scope

- Variable defined within a function is called local scope
- it is accessable within a function only

In [20]:
```
def func():
    local = 90
    print(local)

func()
```

```
90
```

In [21]:
```
print("It will show error as it is local var and can't access outside
print(local)
```

```
It will show error as it is local var and can't access outside funct
ion body

---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
Cell In[21], line 2
      1 print("It will show error as it is local var and can't acces
s outside function body")
----> 2 print(local)

NameError: name 'local' is not defined
```

Enclosing Scope / Non Local Scope

- Inner function can access variable of Outer function

In [22]:
```
def outer():
    NL_scope = 69
    def inner():
        print(NL_scope)
    inner()

outer()
```

```
69
```

Built In scope

- Highest Level of scope that includes all built in functions and objects that are available any where in code

In [23]:
```
print(abs(-5))
```

```
5
```

In [24]:
```
print(len([1,2,3,99]))
```

```
4
```

Global Scope

- it is accessable on entire script defined at top level anywhere

In [25]:
```
glob = 10
def func():
    print(glob)

func()
```

```
10
```

Global Keyword

to change `global` varibale globally

In [26]:
```
var = 10
def check():
    global var
    var = 20
    print(var)
check()

print("Global variable changed globally when check() function executed
```

```
20
Global variable changed globally when check() function executed 20
```

In [27]:
```
counter = 0

def increment_counter():
    global counter  # Declare 'counter' as a global variable within th
    counter += 1

def print_counter():
    print("Counter:", counter)

# Increment the counter multiple times
increment_counter()
increment_counter()
increment_counter()

# Print the counter
print_counter()  # Output: Counter: 3
```

```
Counter: 3
```

Decorator

a decorator is a design pattern that allows you to add functionality to an existing function or method without modifying its structure.

Decorators are implemented as functions themselves and are typically used to modify the behavior of other functions or methods.

The basic syntax for using a decorator is to prefix the function or method definition with the "@" symbol followed by the decorator function name.

When the decorated function or method is called, it is passed as an argument to the decorator function, which then modifies its behavior as desired.

In [28]:
```
def my_decorator(func):
    def check():
        func()
        print("Did you Follow me !")
        print("It's Free of cost")
    return check

@my_decorator
def geek():
    print("Hii ! Buddy")

geek()
```

```
Hii ! Buddy
Did you Follow me !
It's Free of cost
```

Closure

- A closure in Python is a nested function that captures and retains the environment in which it was created,
- even after the outer function has finished execution.
- This allows the inner function to access and manipulate variables from the outer function's scope.

In [29]:
```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

# Create a closure by calling outer_function
closure = outer_function(5)

# Use the closure
result = closure(3)
print(result)
```

```
8
```

Generator

- a generator is a special type of iterator that allows you to iterate over a sequence of values without the need to store them all in memory at once.
- Generators are defined using functions and the yield keyword,

- which allows them to generate values one at a time, on-the-fly, as they are requested.
- All generators are Iterators
- `iter()` used to get the object code
- `next()` used to interate through it

In [30]:
```
def generator(n):
    for i in range(n):
        yield i

obj = generator(5)

print(next(obj))
print(next(obj))
print(next(obj))

print(iter(obj))
```

```
0
1
2
<generator object generator at 0x0000023CCB58B920>
```

Iterator

- an iterator is an object that represents a stream of data.
- It allows you to iterate over elements in a sequence one at a time, without having to load the entire sequence into memory at once.
- Iterators are commonly used in for loops, comprehensions, and other constructs that require sequential processing of elements.

An iterator must implement two methods:

- `__iter__()`: This method returns the iterator object itself.
- `__next__()`: This method returns the next item in the sequence.
- If there are no more items, it raises the StopIteration exception.

In [31]:
```
data = [2,3,4,66,7,1]

obj = iter(data)

print(obj)
```

```
<list_iterator object at 0x0000023CCB21AF50>
```

In [32]:
```
print(next(obj))
```

```
2
```

In [33]:
```
while True:
    try:
        print(next(obj))
    except Exception as e:
        break
```

```
3
4
66
7
1
```

Recursion

- Function calling itself

factorial

In [34]:
```
def factorial(no):
    if no==0:
        return 1
    else:
        return no * factorial(no-1)
factorial(3)
```

Out[34]:
```
6
```

fibonacci no

In [35]:
```
def fibinacci(no):
    if no<=1:
        return no
    else:
        return fibinacci(no-1) + fibinacci(no-2)
result=fibinacci(11)
result
```

Out[35]: 89

power calculation

In [36]:
```
def power(t,o):
    if o==0:
        return 1
    else:
        return t* power(t,o-1)
result=power(2,3)
result
```

Out[36]: 8

* args

*** treat arguments as tuple**

args can be any name

In [37]:
```
def func(*args):
    for i in args:
        print(i)
func(1,2,3,67,7,'udaya')
```

```
1
2
3
67
7
udaya
```

**kwargs

treat arguments as dictionary

In [38]:
```
def func(**kwargs):
    for k,v in kwargs.items():
        print(f"{k} {v}")
func(name="ub",age=30,classs="oo")
```

```
name ub
age 30
classs oo
```

both

In [40]:
```
def function(*args,**kwargs):
    for i in args:
        print(i)
    for k,v in kwargs.items():
        print(f"{k} {v}")
function(1,2,name="ub",age=30,classs="oo")
```

```
1
2
name ub
age 30
classs oo
```

Thank You From Udaya