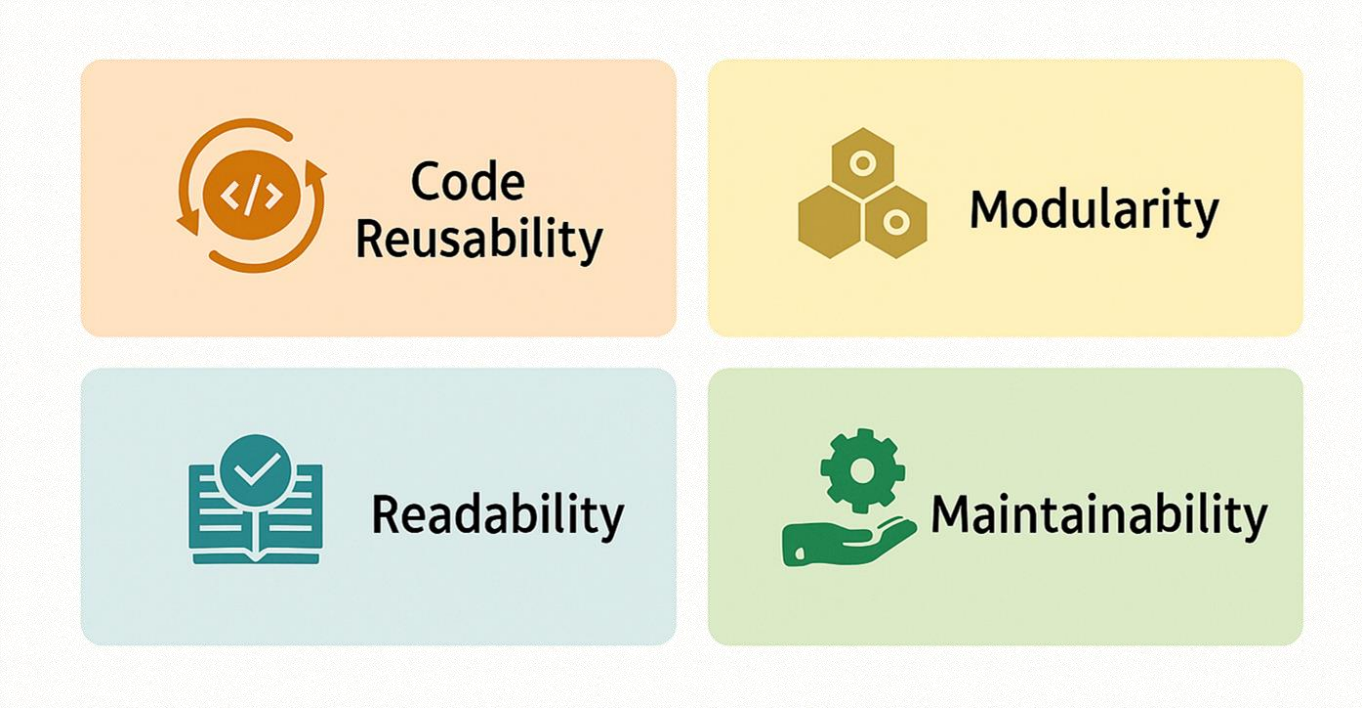
Introduction to Python Language

DEFINING FUNCTIONS

Python Functions

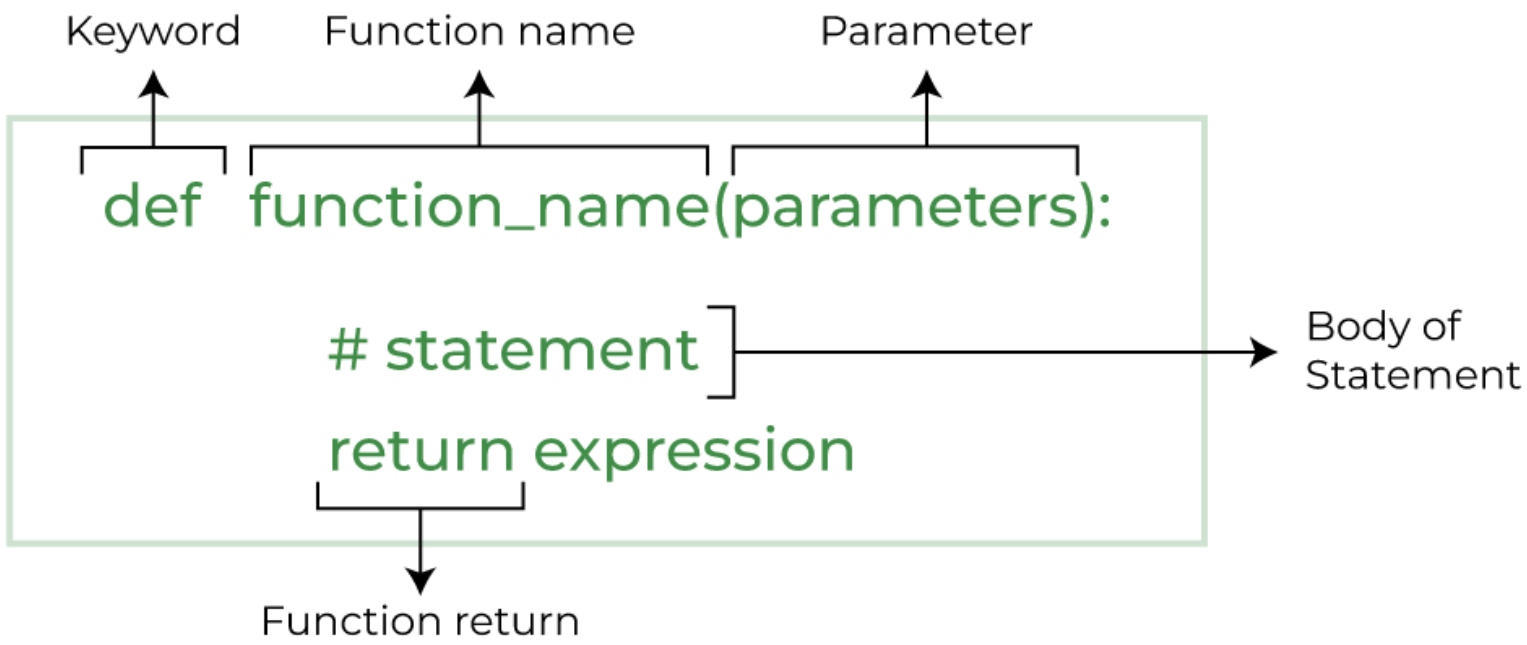


Python functions are groups of statements designed to perform a specific task. They allow us to collect operations that are used frequently or repeatedly into a single reusable unit. Instead of rewriting the same code for different inputs, we can simply call the function and execute the code inside it whenever needed

Defining a Function

- We can define a function in Python, using the def keyword. A function might take input in the form of parameters.
- The syntax to declare a function is:

Syntax:



- **Example:** Here, we define a function using def that prints a happy birthday message when called.

```
def happy():
        print("Happy birthday to you!")
```

- **Calling a Function:** Once the function created, it can called by using the name of the function followed by parenthesis containing parameters of that particular function.
- **Example:**

```
>>> happy() # Driver code to call a function
>>> Happy birthday to you!
```

- **Example:** Let's improve our example by adding the rest of the song for Bilge.

Program script

```
def bdayBilge():
        happy()
        happy()
        print("Happy birthday, dear Bilge")
        happy()
```

Console

```
>>> bdayBilge()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Bilge
Happy birthday to you!
```

Example: Let's improve further by including the person name as parameter of our function. Thus, our function will work properly for anybody.

Program script

formal-parameter

```
def bday(person):
        happy()
        happy()
        print("Happy birthday dear,", person+ ".")
        happy()
```

Console

actual-parameters

Console

```
>>> bday("Fatih")
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fatih.
Happy birthday to you!
```

```
>>> bday("Suleyman")
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Suleyman.
Happy birthday to you!
```

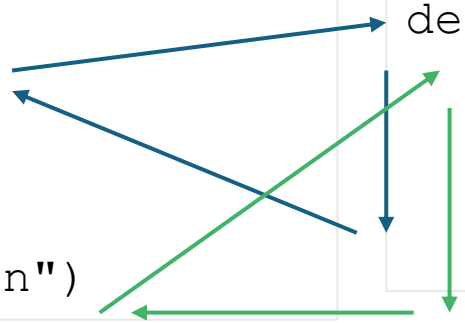
- When Python comes to a function call, it initiates a four-step process:

1. The calling program suspends execution at the point of the call.
2. The formal parameters of the function get assigned the values supplied by the actual parameters in the call.
3. The body of the function is executed.
4. Control returns to the point just after where the function was called.

```
def main():
        bday("Fatih")

        print()

        bday("Suleyman")
```



```
def bday(person):
        happy()
        happy()
        print("Happy birthday dear,", person+ ".")
        happy()
```

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fatih.
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fatih.
Happy birthday to you!
```

More on Function Arguments

- Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

Example: Let's write a function that prompts whether the values passed from keyboard are odd or even.

```
def oddeven(x):
        if x%2==0:
                print("Even")
        else:
                print("Odd")
def main():
        content=input("Enter a value: ")
        num=int(content)
        oddeven(num)
main()
```

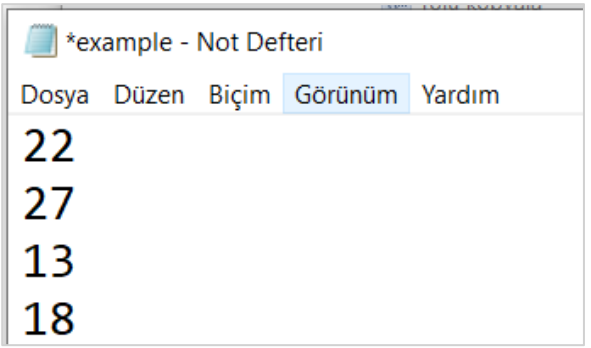
Output:

```
>>>main()
   Enter a value: 12
   Even
```

Example: Let's previous example by reading the values from a file . *Main program will read the values from the file and pass it to the function then the function will prompt «even» or «odd».*

```
def oddeven(x):
    if x%2==0:
        print("Even")
    else:
        print("Odd")

def main():
    with open ("example.txt","r",encoding="utf-8") as f:
        content=f.read() #values acquired by read()
        for i in content.split():
            oddeven(int(i))
main()
```



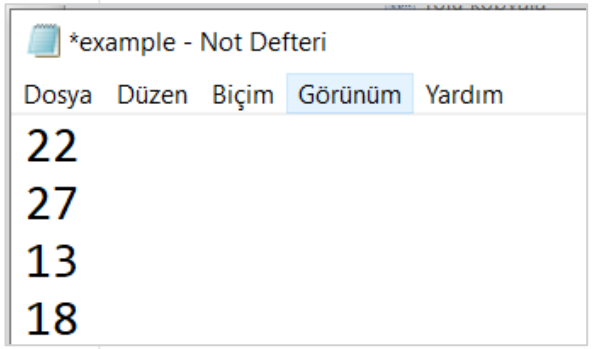
Output:

```
Even
Odd
Odd
Even
```

- Same example can also be in this form without `print()` function

```
def oddeven(x):
    if x%2==0:
        return "Even"
    else:
        return "Odd"

def main():
    with open ("example.txt","r",encoding="utf-8") as file:
        content=file.read()
        for i in content.split():
            oddeven(int(i))
main()
```



Output:

```
Even
Odd
Odd
Even
```

Types of Function Arguments

- Python supports various types of arguments that can be passed at the time of the function call.
- In Python, we have the following function argument types in Python.

1. Default Arguments: A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.

```
def myFun (x, y=50):
        print ("x: ", x)
        print ("y: ", y)
myFun(10)
myFun(10,70)
```

Output:

```
x:  10
y:  50
x:  10
y:  70
```

2. Keyword Arguments: In keyword arguments, values are passed by explicitly specifying the parameter names, so the order doesn't matter.

```
def student(fname, lname):
        print(fname, lname)

student(fname='Mehmet', lname='The Conquirer')
student(lname='The Conquirer', fname='Mehmet')
```

Output:

```
Mehmet The Conquirer
Mehmet The Conquirer
```

3. Positional Arguments: In positional arguments, values are assigned to parameters based on their order in the function call.

```
def nameAge (name, age):
        print("Hi, I am", name)
        print("My age is ", age)

print("Case 1:")
nameAge("Fatih", 21)

print("Case 2:")
nameAge(21, "Fatih")
```

Output:

```
Case-1:
Hi, I am Fatih
My age is  21

Case-2:
Hi, I am 21
My age is Fatih
```

4. Arbitrary Arguments: In Python Arbitrary Keyword Arguments, *args and **kwargs can pass a variable number of arguments to a function using special symbols.

These features provide great flexibility when designing functions that need to handle a varying number of inputs.

There are two special symbols:

***args** in Python (Non-Keyword Arguments)

****kwargs** in Python (Keyword Arguments)

```
# *args example
def fun (*a):
        return sum(a)
print( fun(5,10,15)) # calling and printing results
```

Output:

```
30
```

```
# *args example
def myfun (*arg):
        for a in arg:
                print(a)

myfun('Hello','Welcome','Mech. Eng.')
```

```
Hello
Welcome
Mech. Eng.
```

`def myFun(*arg):` defines a function that accepts any number of positional arguments.
`for a in arg:` loops through the tuple of arguments.

Example: Let’s use *args to multiply any number of values.

```
def multiply (*args):
        result=1
        for a in args:
                result*=a
return result

print( multiply(2,3,4)) # calling and printing results
```

Output:

```
24
```

Keyword Arguments : **kwargs allows us to pass any number of keyword arguments (arguments in the form key=value). These arguments are collected into a **dictionary**, where:

- Keys = argument names
- Values = argument values

```
# *kwargs example
def fun (**kw):
        for k, val in kw.items():
            print(k, val)
fun(a=1,b=2,c=3)) # calling and printing results
```

Output:

```
a 1
b 2
c 3
```

```
# *kwargs example
def introduce(**kw):
       details=[]
       for k, val in kw.items():
           details.append(k+ ':'+ str(val))
       return ', '.join(details)

#calling and printing results
print(introduce(Name='Ali', Age=25, City=Adana)
```

Output:

```
Name:Ali, Age=25, City=Adana
```

```python
# *kwargs example
def introduce(**kw):
        details=[]
        for k, val in kw.items():
                details.append(k+ ':'+ str(val))
        return ', '.join(details)

#calling and printing results
print(introduce(Name='Ali', Age=25, City=Adana)
```

Output:

```
Name:Ali, Age=25, City=Adana
```

EXPLANATION:

`def introduce(**kw)` : accepts flexible keyword arguments .

`for k, val in kw.items()` : loop through each key-value pair.

`details.append(k+ ':'+ str(val))` : format each pair as `key:value` and add to list.

`', '.join(details)` : join list items into a single string separated by commas.

Using both *args and **kwargs :

We can also combine *args and **kwargs in the same function. This way, the function can accept both positional and keyword arguments at once.

```
def student_info(*args,**kwargs):
        print("Subjects:", args) #positional arguments
        print("Details:", kwargs) #keyword arguments

# Passing subjects as *args and details as **kwargs
student_info("Math", "Science", "English", Name="Alice", Age=20, City="New York")
```

Output:

```
Subjects: ( 'Math', 'Science', 'English')
Details: {'Name': 'Alice', 'Age': 20, 'City': 'New York'}
```

Explanation:

- ***args:** collects positional arguments into a tuple.
- ****kwargs:** collects keyword arguments into a dictionary.

We can format the output of the previous example.

```
def student_info(*args,**kwargs):
        print("Subjects:")
        for arg in args:
                print(arg, end=";")
        print("\nDetails:")
        for key, value in kwargs.items():
                print(f"{key} = {value}", end=" ")

# Passing subjects as *args and details as **kwargs
student_info("Math", "Science", "English", Name="Alice", Age=20, City="New York")
```

Output:

```
Subjects:

Math;Science;English;

Details:

Name = Alice Age = 20 City = New York
```

Function within Functions

- A function defined inside another function is called an inner function (or nested function). It can access variables from the enclosing function's scope and is often used to keep logic protected and organized.

```
def f1 ():
        s="I love Python "
        def f2 ():
                print (s)
        f2()
f1()
```

Output:

```
I love Python
```

Anonymous Functions

- In Python, an anonymous function means that a function is without a name. As we already know the **`def`** keyword is used to define the normal functions and the **`lambda`** keyword is used to create anonymous functions.

```
lambda arguments : expression
```

- **lambda:** The keyword to define the function.
- **arguments:** A comma-separated list of input parameters like in a regular function.
- **expression:** A single expression that is evaluated and returned.

```
def cube (x): return x*x*x #without lambda
cube_l= lambda x : x*x*x   #with lambda

print (cube(7))
print (cube_l(7))
```

Output:

```
343
343
```

Details on:
Anonymous Functions
Lambda Functions

Example: In the example, we defined a lambda function to convert a string to its upper case using `upper()`.

```
s1= "Gaziantep"
s2= lambda func : func.upper()    #with lambda

print (s2(s1))
```

Output:

```
GAZİANTEP
```

Using Lambda Function with Condition Checking

A lambda function can include conditions using if statements. Here, the lambda function uses nested if-else logic to classify numbers as Positive, Negative or Zero.

```
n= lambda x : "Positive" if x>0 else "Negative" if x<0 else "Zero"

print (n(5))
print (n(-3))
print (n(0))
```

Output:

```
Positive
Negative
Zero
```

Using Lambda Function for Returning Multiple Results: Here, the lambda calculates both sum and product of two numbers and returns them as a tuple.

```
calc= lambda x,y : (x+y,x*y)

print (calc(3,4))
```

Output:

```
(7,12)
```

Using Lambda Function Using with `filter()`: filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence "sequence", for which the function returns True.

Here, the lambda is used as a filtering condition to keep only even numbers from the list.

```
n=[1,2,3,4,5,6]
even= filter( lambda x: x%2==0, n)

print (list(even))
```

Output:

```
[2,4,6]
```

Details on: `filter()`

Return Statement in Function

- The **`return`** statement ends a function and sends a value back to the caller.
- It can return any data type, multiple values (packed into a tuple), or None if no value is given.

```
def square_value (x):
        return x*x

print (square_value(2))
print (square_value(-4))
```

Output:

```
4
16
```

Pass by Value and Pass by Reference

In Python, variables are references to objects. When we pass them to a function, the behavior depends on whether the object is mutable (like lists, dictionaries) or immutable (like integers, strings, tuples).

- **Immutable objects:** Immutable objects such as integers, strings, and tuples cannot be changed in place. When passed to a function, Python behaves *as if* it is passing by value.

```
def modify_number(x):
    print("Inside function, before change:", x)
    x = x + 10
    print("Inside function, after change:", x)

num = 5
modify_number(num)
print("Outside function:", num)
```

```
Inside function, before change: 5
Inside function, after change: 15
Outside function: 5
```

- **Mutable objects:** Mutable objects such as lists, dictionaries, and sets can be changed in place. When passed to a function, modifications affect the original object.

```
def modify_list(lst):
    print("Inside function, before change:", lst)
    lst.append(100)
    print("Inside function, after change:", lst)

numbers = [1, 2, 3]
modify_list(numbers)
print("Outside function:", numbers)
```

```
Inside function, before change: [1, 2, 3]
Inside function, after change: [1, 2, 3, 100]
Outside function: [1, 2, 3, 100]
```

Next Lecture

Practicing on Python