Python

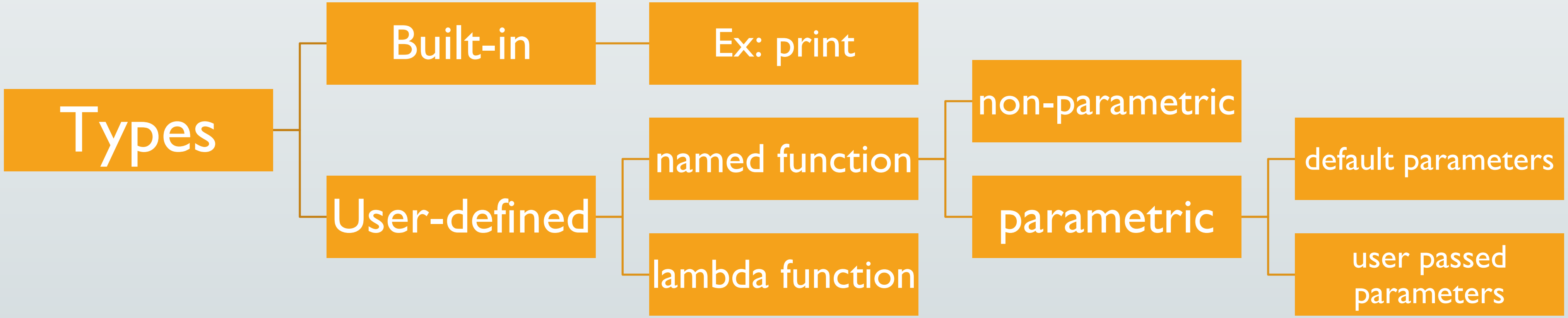
User Defines Functions – Named functions & lambda functions

By-Rageeni Sah

sahrageeni4@gmail.com

Functions

- A group of codes that perform specific task
- A function may take input
- Executes when when it is called
- A function definition can not be empty
- If for some reason it’s empty, use ‘pass’ or ‘return’
- Any variable inside a function has local score



- Types
  - Built-in
    - Ex: print
  - User-defined
    - named function
      - non-parametric
      - parametric
        - default parameters
        - user passed parameters
    - lambda function

Syntax of user-defined-function(named function)

```
## function definition <define function before calling position>
 def <function_name>(parameters):
          statement(s)
          return  # optional


## function call
<function_name>(arguments)
```

Note: Parameters, arguments are optional

Advantages of using functions

- Keeps the program organized, easy to understand and makes it usable
- Reduce redundancy
- enhance readability of the program
- Can be tested individually

Example
# WAP to print a message 4 times

- Without function

```
Print(‘Hello Everyone.’)
Print(‘Hello Everyone.’)
Print(‘Hello Everyone.’)
Print(‘Hello Everyone.’)
```

- With function
- # function definition

```
def greet():
    Print(‘Hello Everyone.’)
    # calling function
    greet()
    greet()
    greet()
    greet()
```

Non-parametric function

- Does not take any parameter

```
# function definition
def greet():
    Print(‘Hello Everyone.’)
# calling function
    greet()
```

Parametric functions

```
# function definition
def greet(name):
    Print(‘Hello’,name,’!’)
# calling function
    greet(‘Peter’)
```

```
#### add two numbers
def add(number1,number2):
        print(number1+number2)
## function call
add(2,3)
add(4,5)
```

Based on the parameters, a function can have any of these parameters

- Named parameters (single/multiple)
- Default parameters
- non-keyworded, variable-length parameters
- Keyworded variable length parameters

Default parametric functions

```
# function definition

def greet(name, str1 = ‘Hello’):

    Print(srt1,name,’!’)

# calling function

    greet(‘Peter’)

    greet(‘Good Morning’,’Peter’)
```

```
#### add a number to existing number

def add(number1,number2=5):

        print(number1+number2)

## function call

add(2,3)

add(5)
```

Note: here sequence of the parameters are imperative
Mandatory parameters should come first in the sequence during function definition

Multiple parameters

```
## WAP to numbers.
def add():
   <logic>
    pass
## call
add(2,3)
add(2,3,4)
add(2,3,4,5)
```

*agrs

- *agrs is pass a non-keyworded, variable-length argument list
- The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args
- With **args*, any number of extra arguments can be tacked on to your current formal parameters
- For example : we want to make a multiply function that takes any number of arguments and able to multiply them all together. It can be done using *args

*kwagrs

- *kwagrs is pass a keyworded, variable-length argument list
- A keyword argument is where you provide a name to the variable as you pass it into the function
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn’t seem to be any order in which they were printed out

Return statement

- Although return keyword is optional, this keyword is ’must’ to return single or multiple parameters back to the calling function

```
def my_function(x):
  return 5 * x


print(my_function(3))
print(my_function(5))

my_var = my_function(9)
print(my_var )
```

Returning multiple values

```
def my_function(x):
    return (x,5 * x)  ## multiple values are returned using parentheses

my_var = my_function(9)  ### my_var  is  a tuple
var1, var2 = my_function(9)
```

Hands-on

#WAF to find factorial of a number.

# WAF to design calculator program using function.

Lambda/Anonymous functions

Syntax: lambda arguments : expression

Highly efficient approach whenever, a function call is required only once

```
#### function definition
def product_ (x):
        return lambda n:x**n
#### function call
fobj = product_(4)
Print(fobj(3)) ## output = 12
Print(fobj(2)).   ## output = 8
```

Filter using lambda function

Filter function helps to apply a filter on given sequence of numbers

## filter syntax

Filter(function, iterables)

Example:

Filter(lambda x: x%2==0, range(10))### output [0,2,4,6,8]

map using lambda function

map function helps to transform a given sequence of numbers

## filter syntax

map(function, iterables)

Example:

map(lambda x: x**2, range(10)) ### output [0,1,2,9,16…]

reduce using lambda function

Reduce function does help to perform task on the given python data type

## filter syntax

reduce(function, iterables)

Example:

From functools import reduce

reduce(lambda x,y: x+y, range(10)) ###

Hands-on

# WAF to reverse a string.

# WAF to swap two numbers.

# WAF to sort given list.

# Write a Python function to check whether a number is perfect or not.

In number theory, a perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself (also known as its aliquot sum). Equivalently, a perfect number is a number that is half the sum of all of its positive divisors (including itself).
Example : The first perfect number is 6, because 1, 2, and 3 are its proper positive divisors, and 1 + 2 + 3 = 6. Equivalently, the number 6 is equal to half the sum of all its positive divisors: (1 + 2 + 3 + 6) / 2 = 6. The next perfect number is 28 = 1 + 2 + 4 + 7 + 14. This is followed by the perfect numbers 496 and 8128.