10. Functions

April 25, 2023

1 Introduction

- A function is a block of code which only runs when it is called and carries out some specific, well-defined task.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- In Python a function is defined using the `def` keyword

1.1 Creating Function

[1]:
```
# Function to print "Hello World"
def hello_world():
    print("Hello World")
    print("Good Morning")
```

1.2 Calling the function

[2]:
```
hello_world()
```

```
Hello World
Good Morning
```

1.3 Example

- Write a function to find whether the given number is Armstrong number or not Armstrong number is a number that is equal to the sum of the cubes of its own digits.

[3]:
```
def armstrong_number():
    num = int(input("Enter a number: "))
    value = 0

    # find the sum of the cube of each digit
    temp = num
    while temp > 0:
        digit = temp % 10
        value = value +  digit ** 3
        temp = temp // 10
    # display the result
    if num == value:
        print(num,"is an Armstrong number")
    else:
        print(num,"is not an Armstrong number")
```

[4]:
```
armstrong_number()
```

```
Enter a number:  370

370 is an Armstrong number
```

2 Arguments

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, separated with a comma.
- Arguments are also known as **Parameters**

2.1 Example

- Write a program Find out whether the given number is Armstrong number or not Armstrong number is a number that is equal to the sum of the cubes of its own digits.
- Write a function to calculate Armstrong Number, pass the number to this function to analyze.

[5]:
```
def armstrong_number1(num):
    value = 0

    # find the sum of the cube of each digit
    temp = num
    while temp > 0:
        digit = temp % 10
        value = value +  digit ** 3
        temp = temp // 10

    # display the result
    if num == value:
        print(num,"is an Armstrong number")
    else:
        print(num,"is not an Armstrong number")
```

[6]:
```
armstrong_number1(370)
```

```
370 is an Armstrong number
```

2.2 Number of Arguments

- By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
[7]: # For example:
     # Function to print first name and last name together

     def my_function(fname, lname):
         print(fname + " " + lname)
```

```
[8]: # Passing actual number of arguments
     my_function("Jon", "Snow")
```

```
Jon Snow
```

```
[9]: # Passing less arguments than actual
     my_function("Jon")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_16250/2944284149.py in <module>
      1 # Passing less arguments than actual
----> 2 my_function("Jon")

TypeError: my_function() missing 1 required positional argument: 'lname'
```

```
[10]: # Passing more arguments than actual
      my_function("Jon", "Snow","King")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_16250/3692389012.py in <module>
      1 # Passing more arguments than actual
----> 2 my_function("Jon", "Snow","King")

TypeError: my_function() takes 2 positional arguments but 3 were given
```

2.3 Arbitrary Arguments *args

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a tuple of arguments, and can access the items accordingly

```
[11]: # For Example
      # Write a function to list the count and titles of books you got.

      def my_books(*books):
          print("I have {0} books".format(len(books)))
```

```
    print("Following are their names:")
    for i in books:
        print('\t', i)
```

[12]:
```
my_books("A Game of Thrones", "War and Peace")
```

```
I have 2 books
Following are their names:
         A Game of Thrones
         War and Peace
```

[13]:
```
my_books("A Tale of Two Cities", "The Stranger", "Hamlet", "Harry Potter and␣
 ↪the Chamber of Secrets")
```

```
I have 4 books
Following are their names:
         A Tale of Two Cities
         The Stranger
         Hamlet
         Harry Potter and the Chamber of Secrets
```

2.4 Keyword Arguments

- Arguments can also be defined with the `key = value` syntax.
- This way the order of the arguments does not matter.

[14]:
```
# For Example
# Write a function to print personal information of a employee

def emp_info(name, age, gender):
    print("Employee name: " + name)
    print("Age: " + str(age))
    print("Gender: "+ gender)
```

[15]:
```
emp_info(age = 30, name="Rohit", gender="Male" )
```

```
Employee name: Rohit
Age: 30
Gender: Male
```

[16]:
```
emp_info("Rohit", "Male")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_16250/1111462853.py in <module>
----> 1 emp_info("Rohit", "Male")
```

```
TypeError: emp_info() missing 1 required positional argument: 'gender'
```

2.5 Arbitrary Keyword Arguments `**kwargs`

- If you do not know how many keyword arguments that will be passed into your function, add two asterisk `**` before the parameter name in the function definition.
- This way the function will receive a dictionary of arguments, and can access the items accordingly

[17]:
```
# For Example
# Write a function to print information of a employee

def emp_details(**emp_info):
    for i in emp_info:
        print(i,':',emp_info[i])
```

[18]:
```
emp_details(name="Rohit", age="30", department="Development",␣
 ↪Expertise="Python")
```

```
name : Rohit
age : 30
department : Development
Expertise : Python
```

2.6 Default Parameter Value

- Mention the argument value in the function definition itself
- If we call the function without argument, it uses the default value.

[19]:
```
# For Example
# Write a function to print the name of city you belong

def my_city(city="Bangalore"):
    print("I am from", city)
```

[20]:
```
my_city()
```

```
I am from Bangalore
```

[21]:
```
my_city("Mumbai")
```

```
I am from Mumbai
```

3 Return Values

- To let a function return a value, use the `return` statement.
- Statements after return statement are not executed

```
[22]: # For example
      # Function to return cube of given number

      def cube(num):
          cu = num ** 3
          return cu
```

```
[23]: cube(9)
```

```
[23]: 729
```

```
[24]: nine_cube = cube(9)
```

```
[25]: nine_cube
```

```
[25]: 729
```

3.1 Example

- Write a program to find whether the given number is Armstrong number or not Armstrong number is a number that is equal to the sum of the cubes of its own digits.
- Write a function to calculate Armstrong Number,pass the number to this function to analyze.
- This function returns `True` if given number is Armstrong number, else `False`

```
[26]: def armstrong_number2(num):
          value = 0

          # find the sum of the cube of each digit
          temp = num
          while temp > 0:
              digit = temp % 10
              value = value +  digit ** 3
              temp = temp // 10

          # return the result
          if num == value:
              return True
          else:
              return False
```

```
[27]: a = armstrong_number2(370)
```

```
[28]: a
```

```
[28]: True
```

4 Recursion

- Recursion means that a function calls itself.

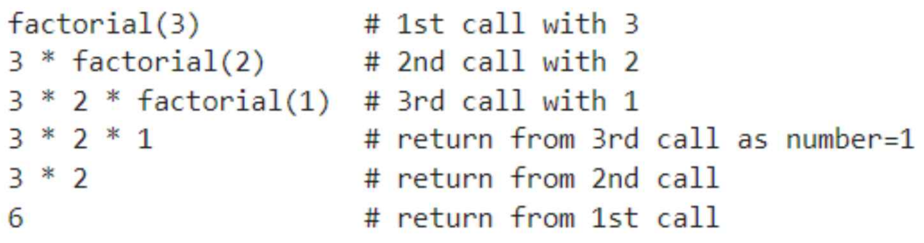
[29]:
```
# For Example
# Function to find factorial of given number

def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
```

[30]:
```
num = 3
factorial(num)
```

[30]:
```
6
```

- Explanation for `factorial(3)`



```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```

- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.
- By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in `RecursionError`

[31]:
```
# RecursionError Example

def recursor():
    recursor()
```

[32]:
```
recursor()
# This might fail in jupyter notebook, for required results run on terminal
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
/tmp/ipykernel_16250/293436695.py in <module>
----> 1 recursor()
      2 # This might fail in jupyter notebook, for required results run on␣
 ↪terminal
```

```

/tmp/ipykernel_16250/2714032471.py in recursor()
      2
      3 def recursor():
----> 4     recursor()

… last 1 frames repeated, from the frame below …

/tmp/ipykernel_16250/2714032471.py in recursor()
      2
      3 def recursor():
----> 4     recursor()

RecursionError: maximum recursion depth exceeded
```

4.1 Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

4.2 Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

5 Docstring

- Documentation strings (or docstrings) provide a convenient way of associating documentation with functions, classes, and methods.
- The docstring should describe what the function does, not how.
- **Declaring Docstrings:** The docstrings are declared using `'''`triple single quotes`'''` or `"""`triple double quotes`"""` just below the class, method or function declaration.
- Accessing Docstrings: The docstrings can be accessed using the `__doc__` method of the object or using the `help` function.

[33]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(…)
    print(value, …, sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

[34]:
```python
# For Example
# function to find whether the given number is Armstrong number or not
def armstrong_number3(num):
    '''Function to find whether the given number is Armstrong number or not.'''
    value = 0

    # find the sum of the cube of each digit
    temp = num
    while temp > 0:
        digit = temp % 10
        value = value +  digit ** 3
        temp = temp // 10

    # return the result
    if num == value:
        return True
    else:
        return False
```

[35]:
```python
help(armstrong_number3)
```

```
Help on function armstrong_number3 in module __main__:

armstrong_number3(num)
    Function to find whether the given number is Armstrong number or not.
```

[36]:
```python
armstrong_number3.__doc__
```

[36]:
```
'\n    Function to find whether the given number is Armstrong number or not.\n
'
```

What should a docstring look like?

- The doc string line should begin with a capital letter and end with a period.
- The first line should be a short description.
- If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description.
- The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

6 Docstring Format

[ ]:
```python
def add_nums(num1, num2):
    """Add up two integer numbers.

    This function simply wraps the ``+`` operator, and does not
    do anything interesting, except for illustrating what
    the docstring of a very simple function looks like.

    Args:
        num1 (int) : First number to add.
        num2 (int) : Second number to add.

    Returns:
        int: The sum of ``num1`` and ``num2``.

    Raises:
        AnyError: If anything bad happens.
    """
    return num1 + num2
```

7 Type Hinting

Type hinting is a formal solution to statically indicate the type of a value within your Python code. It was introduced in Python 3.5.

[ ]:
```python
def greet(name: str) -> str:
    return "Hello, " + name
```

8 Anonymous Function

- An anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the `lambda` keyword.
- Hence, anonymous functions are also called Lambda functions.

[38]:
```python
# find square of numbers using lambda functions
square = lambda x: x ** 2
```

[39]:
```python
square(10)
```

[39]:
```
100
```

9 pass Statement

- Function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.
- pass statement also applies to conditional statements (if, else, elif)

[40]:
```
def myfunction():
    pass
def get_data():
    pass
def post_data():
    pass
```

[41]:
```
myfunction()
```