intro_to_functions

February 5, 2025

1 Introduction to Python Functions

In this notebook, we'll explore what functions are, why they are useful, and how to create and use them in Python. We'll use examples relevant to data analytics to make the concepts more concrete.

1.1 What is a Function?

A function is a reusable block of code that performs a specific task. Functions allow you to: - Break down complex problems into smaller, manageable pieces. - Reuse code without rewriting it. - Improve code readability and organization.

In Python, functions are defined using the def keyword, followed by the function name and parentheses ().

1.2 Why Use Functions?

- Reusability: Write code once and use it multiple times.
- Modularity: Divide your program into separate, manageable sections.
- Maintainability: Easier to update and fix code.
- Abstraction: Hide complex details and expose simple interfaces.

1.3 Defining a Function in Python

Here's the basic syntax for defining a function:

def function_name(parameters): “ “ “ Optional function documentation (docstring) ” “ “ # Function body return result

- def: Keyword to define a function.
- function_name: Name of the function (should be descriptive).
- parameters: Optional inputs the function accepts.
- return: (Optional) Keyword to return a value from the function.

1.4 Function Arguments

Functions can accept inputs, known as arguments or parameters, to make them more flexible.

1.4.1 Positional Arguments

Arguments that are passed to a function in order based on their position.

[1]:
```
# Here we define a function called greet that takes two arguments,
# name and message.

def greet(name, message):
    """
    Returns a greeting based on the name and message arguments.
    """
    print(f"{message}, {name}!")
```

[2]:
```
greet("Alice", "Good afternoon")
```

```
Good afternoon, Alice!
```

1.4.2 Keyword Arguments

Arguments can be passed using the parameter names. This can make the code clearer, especially if there are several arguments. It also means that you don't have to use the same order as the parameter list in the function definition.

[3]:
```
greet(message="Good morning", name="Bob")
```

```
Good morning, Bob!
```

1.4.3 Default Arguments

We can specify a default value for parameters.

[4]:
```
def greet(name, message="Hi"):
    """
    This version of the greet function uses a default message of Hi
    """
    print(f"{message}, {name}!")
```

[5]:
```
# Note that we do not specify the value of message when calling greet.
greet("Charlie")
```

```
Hi, Charlie!
```

1.5 Return Statement

The return statement exits a function and allows it to pass back a value.

[6]:
```
def add(a, b):
    """
    Return the result of adding a and b
    """
    return a + b
```

[7]:
```
result = add(5, 3)
```

[8]:
```
print(f"The result of the addition is {result}")
```

```
The result of the addition is 8
```

1.6 Examples

Next we'll explore some examples of more interesting functions. We'll use examples from data analytics. Pandas already provides a lot of functionality such as calculating the mean and dropping rows with missing values, so it will generally be easier to use that.

1.6.1 Calculating the mean value of a list of numbers

[9]:
```
def calculate_mean(data):
    """
    Calculates the mean of a list of numbers.
    """
    n = len(data)
    total = sum(data)
    mean = total / n
    return mean


# Sample data
data = [23, 76, 97, 61, 45, 89, 56]

mean_value = calculate_mean(data)
print(f"The mean is: {mean_value}")
```

```
The mean is: 63.857142857142854
```

1.6.2 Remove 'None' values from a list

[10]:
```
def clean_data(data):
    """
    Removes None values from a list.
    """
    cleaned_data = []
    for item in data:
        if item is not None:
            cleaned_data.append(item)
    return cleaned_data


# Sample data with missing values
raw_data = [10, None, 25, 30, None, 45, 50]

cleaned_data = clean_data(raw_data)
print(f"Cleaned data: {cleaned_data}")
```

```
Cleaned data: [10, 25, 30, 45, 50]
```

1.6.3 Split name

Here we show a function that takes a two-part name like "James Bond" or a three-part name like "Mrs Emma Smith" and returns a tuple of (title, forename, surname). The title will be an empty string in the case of a two-part name.

[11]:
```
def split_name(name):
    """
    Split name into Title, Forename, and Surname. Return these as a tuple.
    Title just gets an empty string if not included in the name.
    """
    parts = name.split()
    if len(parts) == 3:
        return (parts[0], parts[1], parts[2])
    else:
        return ('', parts[0], parts[1])
```

[12]:
```
split_name("Mrs Emma Smith")
```

[12]:
```
('Mrs', 'Emma', 'Smith')
```

[13]:
```
split_name("James Bond")
```

[13]:
```
('', 'James', 'Bond')
```

[14]:
```
split_name("Chewbacca")
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[14], line 1
----> 1 split_name("Chewbacca")

Cell In[11], line 10, in split_name(name)
      8     return (parts[0], parts[1], parts[2])
      9 else:
---> 10     return ('', parts[0], parts[1])

IndexError: list index out of range
```

Oh dear, we've passed in a name that only has one part, but the function assumed that there would be two or three parts.

1.6.4 Challenge

Can you modify the function so that it can deal with one part, two part, or three part names? In the case of a one-part name, like Chewbacca or Madonna, the surname part of the tuple (index 2)

should be an empty string.