8. Functions

Flexible Recycling of Code

What are functions?

Remember `print()`? It’s a **function!**

- Functions are **blocks of code** that perform a specific task
- They allow you to **break down** your **code** into smaller, **reusable** parts
- Functions can take input, perform operations, and return output
- The code within a function can be **flexible** based on the input (→ **parameters**)

Built-in functions

Python comes with a variety of built-in functions that are readily available for use.

Examples of built-in functions include `print(), len(), max(), min(), sum(), abs(), round(), sorted(), range()`, and many more.

To use a built-in function, simply call it by its name followed by parentheses.

Example: `print("Hello, world!")`

Cool thing: we can define our own functions, too!

Defining a function

Parameters = Inputs to the function.
“We’re expecting 2 parameters and we’ll refer to them as `param1` and `param2`”

Write `def` to signal to the interpreter:
“We’re defining a function here!”

```
def function_name(param1, param2):
    # code to be executed
    result = param1 + param2

    return result
```

Function body.
Here comes the code that the function will execute.

This indent space is important!
Leaving the indent = leaving the function body.

Question:
What function name would make sense here?

Using a function

```
def add(param1, param2):
    result = param1 + param2
    return result
```

Now, we can just:

```
mysum = add(3, 7)
print(mysum)
```

Try it!

NOTE

You are ***calling*** the function with the ***arguments*** 3 and 7.

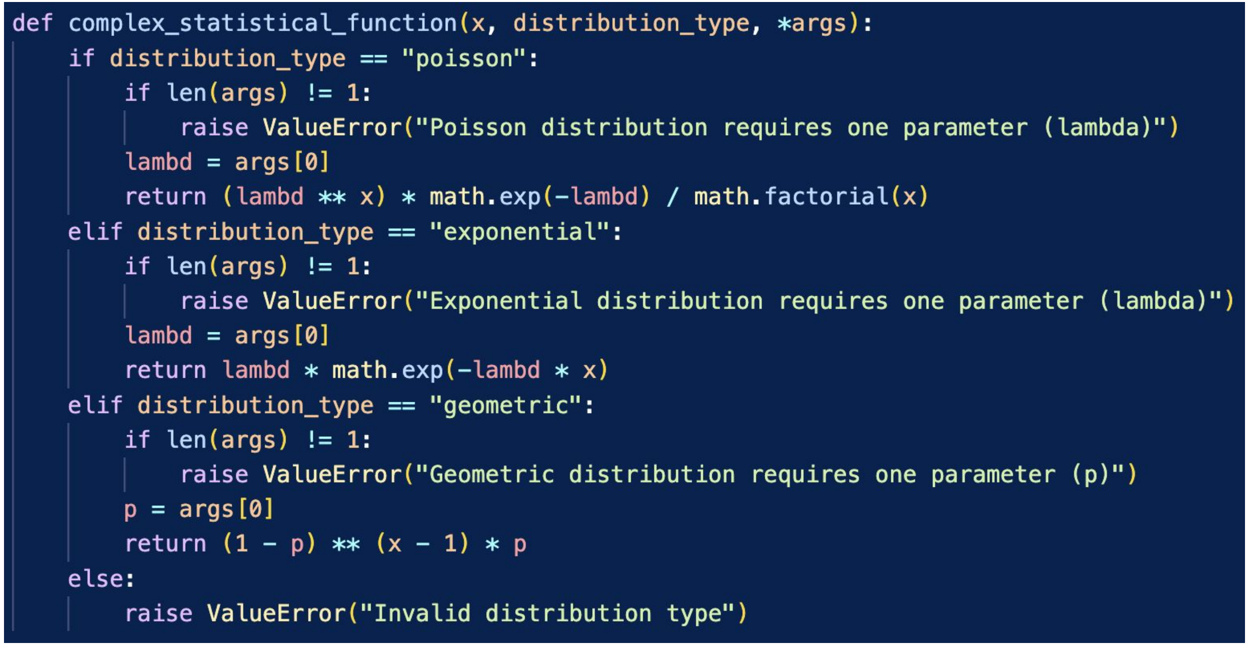
Why though?

```
mysum = add(3, 7)
```

vs.

```
mysum = 3 + 7
```

Imagine our function body is very complicated…



```python
def complex_statistical_function(x, distribution_type, *args):
    if distribution_type == "poisson":
        if len(args) != 1:
            raise ValueError("Poisson distribution requires one parameter (lambda)")
        lambd = args[0]
        return (lambd ** x) * math.exp(-lambd) / math.factorial(x)
    elif distribution_type == "exponential":
        if len(args) != 1:
            raise ValueError("Exponential distribution requires one parameter (lambda)")
        lambd = args[0]
        return lambd * math.exp(-lambd * x)
    elif distribution_type == "geometric":
        if len(args) != 1:
            raise ValueError("Geometric distribution requires one parameter (p)")
        p = args[0]
        return (1 - p) ** (x - 1) * p
    else:
        raise ValueError("Invalid distribution type")
```

Good to define once, and then just reuse by calling the function.

Functions: Exercise

Let’s try to define a function named `subtract` ourselves:

- It should take 2 parameters.
- Inside the function, it should subtract the two.
- Then, return the result.

After you defined it, call the function with some arguments!

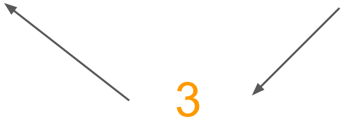
Solution

```
def subtract(a, b):
    res = a - b
    return res


myresult = subtract(4, 1)
print(myresult)
>>> 3
```

Return?

If a function `return`s something, it's "giving" us a value we can "catch".

```
def subtract(a, b):
    res = a - b
    return res


myresult = subtract(4, 1)
```



3

Return vs. Print

Try to just call our function like so: `subtract(4, 1)`

```
def subtract(a, b):
    res = a - b
    return res
```

```
subtract(4, 1)
>>>
```

We didn't "catch" the return

```
print(subtract(4, 1))
>>> 3
```

We "caught" it and gave it to the print() function

No pressure to `return`

A `return` statement isn’t necessary to define a valid function though!
A function can do whatever we want.

```
def subtract(a, b):
    print(“subtracting!”)
    print(a-b)

subtract(10, 7)
>>> “subtracting!”
>>> 3
```

```
def useless_func():
    a = 10


useless_func()
>>>
```

Let’s fix some bugs

```
def add_numbers(a, b):
    sum = a + b


result = add_numbers(5,8)
print(result)
>>>
```

```
def add_numbers(a, b):
    sum = a + b
    return sum

result = add_numbers(5,8)
print(result)
>>> 13
```

Let’s fix some bugs

```
def say_hello():
    print(“Hello”)


say_hello
```

```
def say_hello():
    print(“Hello”)


say_hello()

>>> “Hello”
```

Don’t forget the brackets!

Let’s fix some bugs

```
def division(a, b):
    print(a / b)

# we want to divide 4 by 2
division(2, 4)
>>> 0.5
```

```
def division(a, b):
    print(a / b)

# we want to divide 4 by 2
division(4, 2)
>>> 2
```

Argument order is important!

Let’s fix some bugs

```
def print_text(t):
    print(t)

print_text()

>>> TypeError: print_text()
missing 1 required positional
argument: 't'
```

```
def print_text(t):
    print(t)

print_text(“i was missing!”)

>>> “i was missing!”
```

Common built-in functions

- `print()`: Outputs data to the console.
- `len()`: Returns the length of an object (e.g., a string or list).
- `max()`: Returns the largest item in an iterable.
- `min()`: Returns the smallest item in an iterable.
- `sum()`: Returns the sum of all items in an iterable.
- `abs()`: Returns the absolute value of a number.
- `round()`: Rounds a number to a specified number of decimal places.
- `sorted()`: Returns a new sorted list from the elements of an iterable.
- `range()`: Generates a sequence of numbers.

Let’s get acquainted with common functions

- `len()`: Returns the length of an object (e.g., a string or list).

- Define a list and fill it with elements
- Save the length of the list in a variable `list_len`
- Print `list_len`

```
mylist = [“this”, “is”, “my”, “list”]
list_len = len(mylist)
print(list_len)
>>> 4
```

Let’s get acquainted with common functions

- `max()`: Returns the largest item in an iterable.

- Define a list and fill it with numbers.
- Save the maximum element of the list in a variable `max_element`
- Print `max_element`

```
mylist = [9, 4, 2, 8]
max_element = max(mylist)
print(max_element)
>>> 9
```

Let’s get acquainted with common functions

- `sum()`: Returns the sum of all items in an iterable.

- Define a list and fill it with numbers.
- Save the sum of the list in a variable `list_sum`
- Print `list_sum`

```
mylist = [9, 4, 2, 8]
list_sum = sum(mylist)
print(list_sum)
>>> 23
```

Let’s get acquainted with common functions

- `sorted()`: Returns a new sorted list from the elements of an iterable.

- Define this list: [5, 2, 9, 1]
- Save the sorted version of the list in `sorted_list`
- Print `sorted_list`

```
mylist = [5, 2, 9, 1]
sorted_list = sorted(mylist)
print(sorted_list)
>>> [1, 2, 5, 9]
```

Additional Exercises

Functions

Exercise 1

Write a function `check_value()`, which takes a number as an argument. Using `if` / `else`, the function should print whether the number is bigger, smaller, or equal to 5.

Exercise 2

Write a function `check_length()`, which takes a string as an argument.
Using `if` / `else`, check if the length of the string is bigger, smaller, or equal to 10 characters.

Exercise 3

Write a function `print_numbers()`, which takes a **list of numbers** as argument. Using a `for` loop, print each number one by one.

Solution 3

```
def print_numbers(num_list):
    for element in num_list:
        print(element)
```

Exercise 4

Write a function `check_each()`, which takes a list of numbers as argument. Using a `for` loop, iterate through the list.

For each number, print “bigger” if it’s bigger than 5, “smaller” if it’s smaller than 5, and “equal” if it’s equal to 5.

Exercise 5

Write a function `add_one()`. It takes an integer as argument. The function adds 1 to the integer and returns it.

Write another function `add_one_to_list()`. It takes a list of integers as argument.
Define a variable `new_list` in this function.
Using a for loop, iterate through the argument list.
Using `add_one()`, fill `new_list` with integers from the argument list incremented by 1.
Print `new_list`.

Example:

```
add_one_to_list([1, 2, 3])
>>> [2, 3, 4]
```