

# ! Assignment 1 is this Friday

You **must go to your tutorial** to do the assignment this Friday.



The best way to prepare is to work through the worksheets!

- Assignment questions will be very similar
- Ask any final questions in the first half hour of the tutorial



Bring your laptop!!

- Don't have one? Talk to your SA
- Device problems? Talk to your tutor and then go to AT 6.06 at 5pm to retake it

# > Today

Fiona has talked to you about...

- Built-in functions
  - `print()`, `len()`, etc.
- Built-in types & methods
  - `str`, `int`, `list`, etc.
  - `.upper()`, `.strip()`, etc.

*...today we'll make our own!*

→ Lecture 6 – Functions

→ Lecture 7 – Objects 1: Classes & instances

**INF1-IP**

# Lecture 6 – Functions

Rayo Verweij

12/10/2026



# > What's a function?

Functions are **reusable pieces of code** that can have their own **input, process, and output**.

*For example...*

| Function             | Input                | Process                                                      | Output                                  |
|----------------------|----------------------|--------------------------------------------------------------|-----------------------------------------|
| <code>print()</code> | Any                  | Sends its input to the terminal for display                  | None                                    |
| <code>input()</code> | A string             | Displays the prompt in the terminal and gets resulting input | A string containing input from terminal |
| <code>len()</code>   | String or collection | Counts its length                                            | Input's length as an integer            |
| <code>type()</code>  | Any                  | Determines the input's type                                  | The input's type                        |

# > Python has a bunch built-in!

Including these absolute surprises...

- *min()* returns the minimum of a collection
- *max()* returns the maximum of a collection
- *sum()* returns the sum of a collection

```
scores = [1, 2, 3]
```

```
min(scores)    # 1
```

```
max(scores)    # 3
```

```
sum(scores)    # 6
```

Some terminology: *scores* is the **argument** of the function **calls**

# > Defining our own functions



Thanks Julia!

This is Jimmy! Jimmy "loves food but despises people"

```
# Defining a function named 'bark'
def bark():
    print("Woof!")
    print("Woof!")

# Calling the bark function
bark()
```

**Note:** keyword (**def**) + colon + indentation!

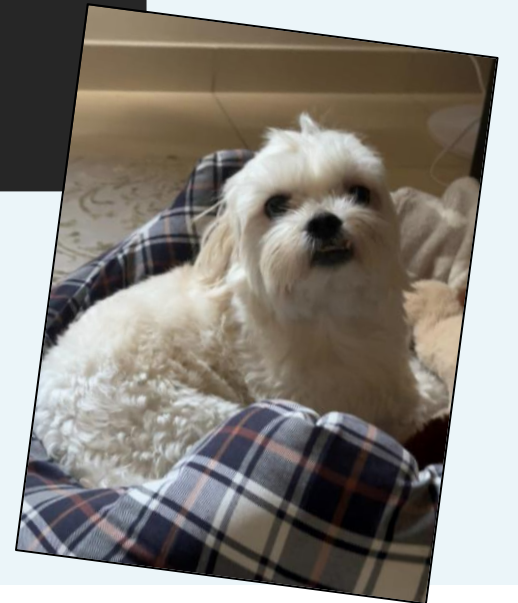
# > Parameters

**Parameters** define which **arguments** a function can take as input.

```
# Defining a function named 'munch' with the parameters 'name' and 'treat'  
def munch(name, treat):  
    print(f"{name} loves to eat some {treat}")
```

```
# Calling the munch function with the arguments 'Jimmy' and 'biscuits'  
munch("Jimmy", "biscuits")
```

→ Jimmy loves to eat some biscuits



# > Returning values

The `return` keyword gives the output of a function back to the code.

```
def how_many_eyes(species):  
    if(species == "human" or species == "dog"):  
        return 2  
    elif(species == "spider"):  
        return 8  
    else:  
        return -1  
  
jimmy_eyes = how_many_eyes("dog")  
print(f"Jimmy has {jimmy_eyes} eyes")
```

→ Jimmy has 2 eyes

# { Try it yourself #1

Write a function checks whether a number is divisible by 3.

If so, return True; if not, return False.

Then test your function with some example values!

# { Example solution

```
def divisible_by_three(num):  
    if(num % 3 == 0):  
        return True  
    else:  
        return False  
  
print(divisible_by_three(5))    # False  
print(divisible_by_three(60))  # True
```

# > What's next?

Well done, these are the basics of writing functions!

Next, some functionality and some peculiarity:

- None
- Keyword arguments
- Default values
- Documenting a function
- Variable scope

# > Nothing to see here

Functions **always return something**.

If you don't use return, or keep it empty, the function will return **None**.

None is a special value that simply exists for this purpose.

```
def nothing():  
    return  
  
print(nothing())
```

→ None

# > Keyword arguments

By default, arguments are **positional**: supply them in parameter order.

You can also, however, explicitly mention the parameters you're supplying.

```
def describe(name, eye_colour, hair_colour):  
    print(f"{name} has {eye_colour} eyes and {hair_colour} hair.")  
  
# Calling describe using *positional* arguments  
describe("Max", "blue", "brown")  
  
# Calling describe using *keyword* arguments  
describe(eye_colour="black", name="Alex", hair_colour="red")
```

→ Max has blue eyes and brown hair

→ Alex has black eyes and red hair

# > Default parameter values

Parameters can be made **optional** by supplying a **default value**.

```
def describe(name, eye_colour, hair_colour="black"):
    print(f"{name} has {eye_colour} eyes and {hair_colour} hair.")

# Calling describe by defining each parameter
describe("Max", "blue", "brown")

# Calling describe with the default hair_colour
describe("Liz", "amber")
```

→ Max has blue eyes and brown hair

→ Liz has amber eyes and black hair

# ! Watch out with keywords and defaults!

Note that the `=` sign has different meanings based on where in the function it's used!

- In the function *call*, it supplies *keyword arguments*
- In the function *declaration*, it defines *default values*

Keep your code tidy:

- Put required parameters before optional ones
- Put positional arguments before keyword ones

## > Multi-line strings

Strings enclosed in **triple quotes** can **span multiple lines**!

```
longwinded = """This is quite  
a long piece of  
text that I'm writing  
here and it's still going  
and going further  
and going on  
never giving up  
never letting down  
like Rick said"""
```

Both single and double quotes work.

It is particularly useful when trying to preserve newlines.

# > Docstrings

Triple-quoted strings are often used to **document functions**.

Calling the built-in `help()` function shows a function's **docstring**.

```
def square(n):  
    """Takes in a number n and returns its square."""  
    return n * n  
  
help(square)
```

*Help on function square in module \_\_main\_\_:*

*square(n)*

*Takes in a number n and returns its square.*

**Note:** only pass the **name** of the function to `help()`, don't call it!

# { Try it yourself #2

Fix the piece of code that's provided in the notebook and provide appropriate documentation.

# { Solution

```
def vowel_count(word, vowels={"a", "e", "i", "o", "u"}):  
    """Count the number of vowels in 'word'. Default set of vowels is {a,e,i,o,u}"""  
    total = 0  
    for letter in word:  
        if letter in vowels:  
            total += 1  
    return total  
  
print(vowel_count("hello"))  
print(vowel_count("alloy", {"a", "e", "i", "o", "u", "y"}))
```

## > Variable scope

Variables defined inside of a function are **local** to that function.  
They can only be used inside!

Variables defined outside of functions are **global** to your program.

This is referred to as the variable's **scope**.

# > Summary

In this lecture, we have...

- Recognised how the IPO pattern is present in Python functions
- Defined and called functions with parameters and return values
- Used keyword arguments and default parameter values
- Documented functions with docstrings
- Understood *None* and variable scope