Functions

MAD 102

Functions/ Modules

- Structured programming is the method for creating programs from sequence, selection and repetition structures and it is about clarity
  - You should **not** have to read the entire program to determine what it does or what each section does
  - Using comments ***consistently*** is one way to make the programs clear.

Functions/ Modules

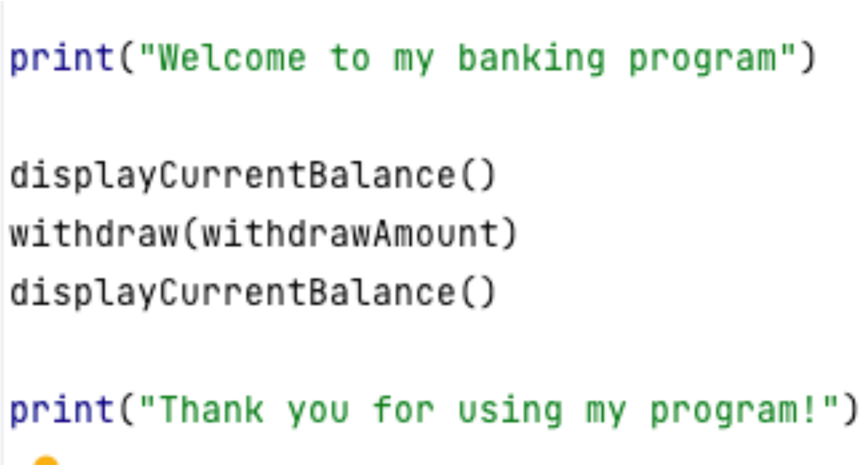
- Another way is to use ***modules* – ** sections of code that ***perform a specific task***
  - Instead of listing all the code in the main program, your main program calls a **module** and anyone who uses the program gets an idea about what the module does *from its **name** and the **accompanying comments***

Top-down design

- ***Top-down design*** is a well-tested means of program design that is :
  - Used to create a program's overall outline
  - Describe the tasks to be accomplished
  - The details of these tasks are refined later
- The program's ***main module*** has until now contained all the program's code
- Additional modules are created and placed outside of the main module and are called when needed

Top-down design

- Example:



```
print("Welcome to my banking program")

displayCurrentBalance()
withdraw(withdrawAmount)
displayCurrentBalance()

print("Thank you for using my program!")
```

- Main module calls the following modules:
  - `displayCurrentBalance(), withdraw(), and displayCurrentBalance()`

Modularizing a program

- Module names follow similar rules as variables:
  - Start with a lowercase letter and can contain letters, digits, hyphens and underscores
  - ***No*** spaces
  - The module name is followed by parentheses to enclose any **arguments** that need to be sent to the module
  - Python best practices implements lowercase lettering for its function names
    - For clarity, use _ between words
      - math_calculations vs mathcalculations

Modularizing a program

- The name should describe what the function does

- **Correct** function name: `calculate_interest`
- **Incorrect** function name:

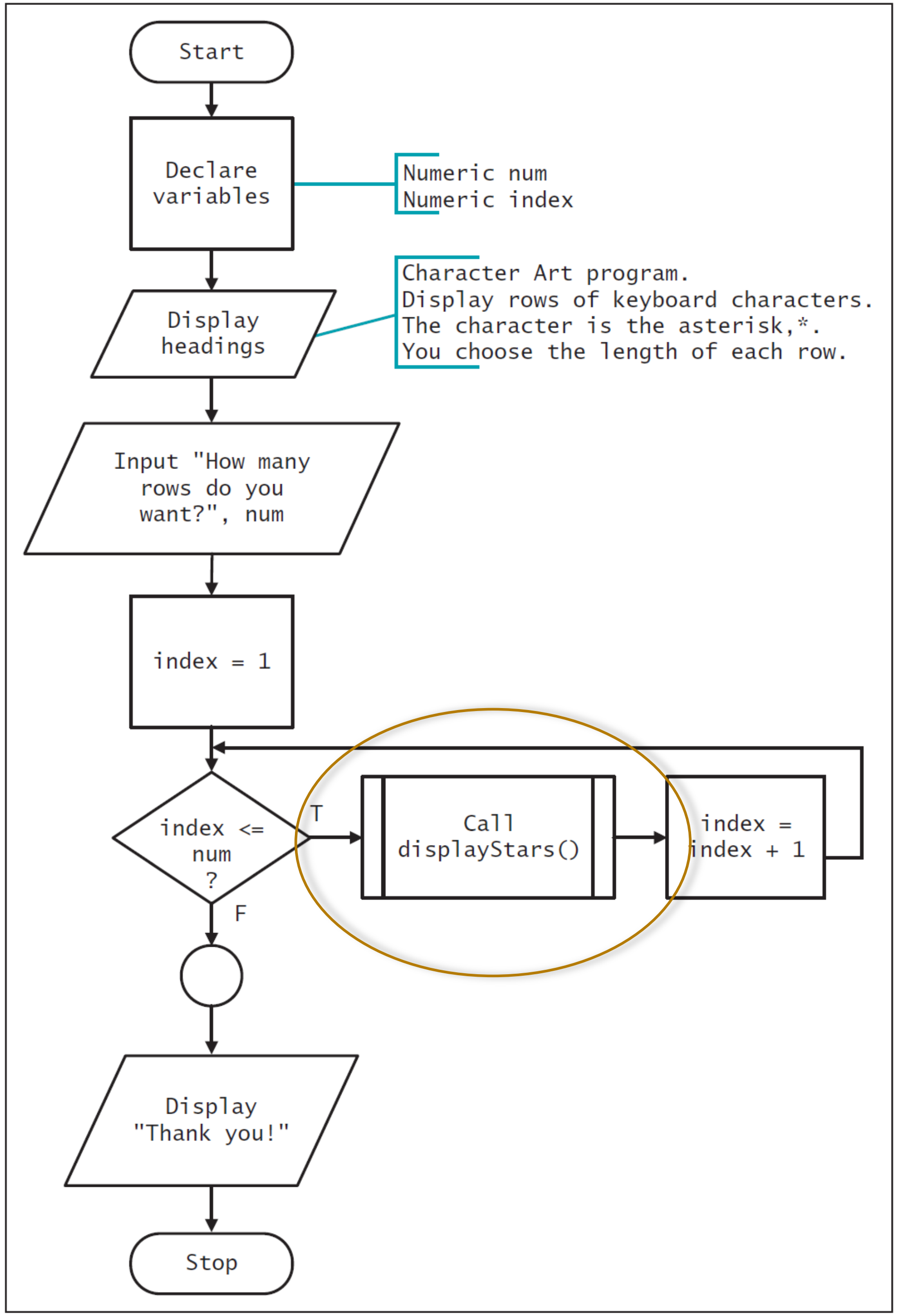
`Calculate_Interest` Or

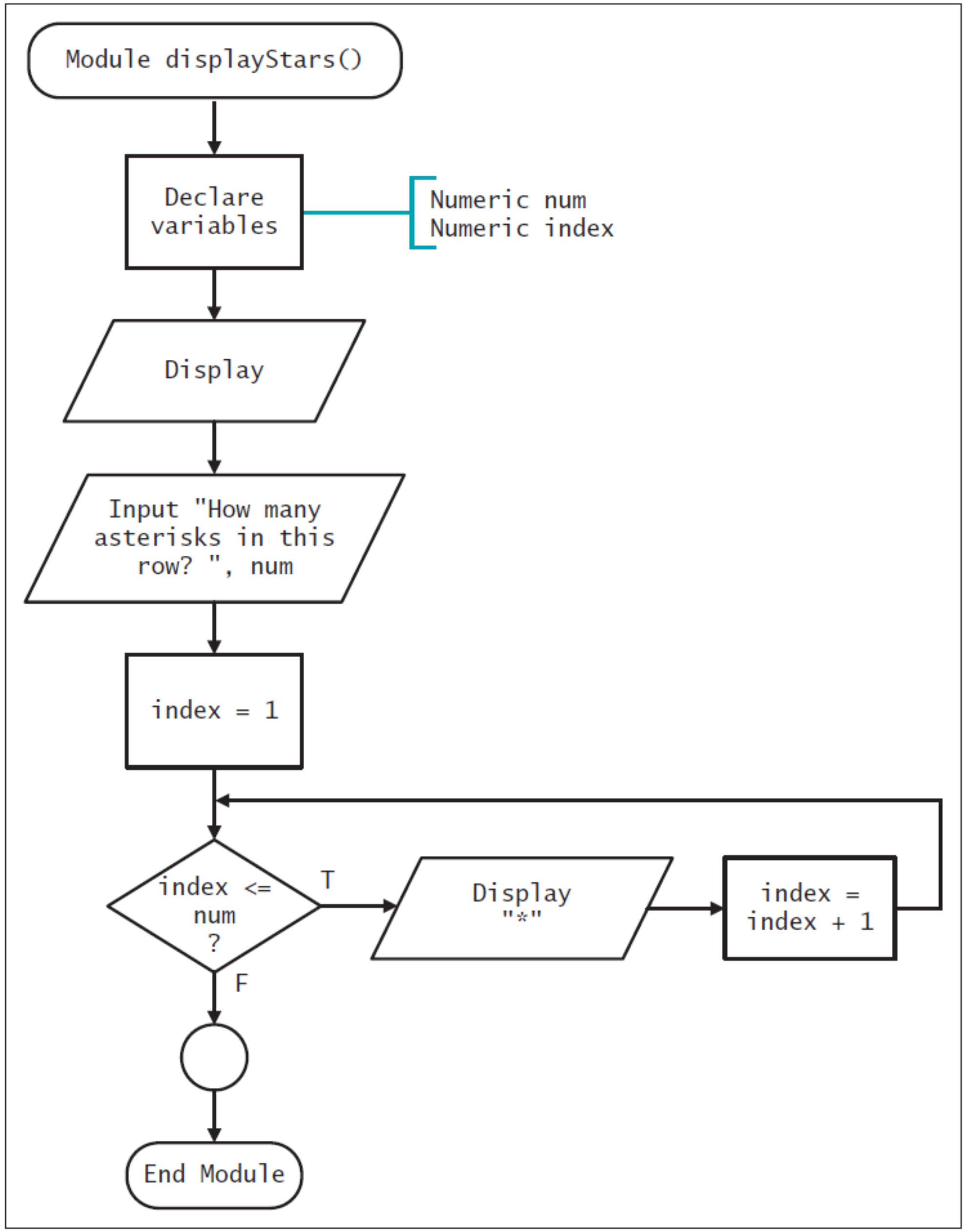
`calculate` Or

`calc`

Flowcharting modules

- The flowchart shape for a module is the module symbol – a rectangle with stripes.
- The text inside contains the command keyword Call followed by the name of the module to call
    - `Call display_stars()`
- The called module has its own flowchart in which the terminating symbols contain the keyword `Module` followed by the name of the module and to end the keywords `End Module`
- Otherwise all other symbols are the same as the main module


Start
Declare variables
Numeric num
Numeric index
Display headings
Character Art program.
Display rows of keyboard characters.
The character is the asterisk,*.
You choose the length of each row.
Input "How many rows do you want?", num
index = 1
index <= num ?
T
Call displayStars()
index = index + 1
F
Display "Thank you!"
Stop


Module displayStars()
Declare variables
Numeric num
Numeric index
Display
Input "How many asterisks in this row? ", num
index = 1
index <= num ?
T
Display "*"
index = index + 1
F
End Module

Functions

- Functions are self-contained chunks of code
  - They perform a specific task – the task may be a repetitive one
  - The **function definition** consists of the function’s name and the block of statements
- Functions have names which are used when you ‘call’ them
  - The code contained in the function is then run

Functions

- Functions are named using names that describe the task the function performs
- Module names follow similar rules as variables:
  - Camel casing is used
  - Start with a lowercase letter and can contain letters, digits, hyphens and underscores
  - ***No*** spaces
  - The module name is followed by parentheses to enclose any arguments that need to be sent to the module

Functions

- Functions define arguments that you can use to pass data that help the function perform its work
- Functions can return something after the work is performed

Functions - Scope

- In many languages, variables that are declared ***inside*** a module are considered to be local to that module
    - This means that they are only available to the code located inside the module (are only available while the module is being performed)
    - When the end module statement is encountered, local variables are no longer in use or accessible to the program
- To make the variable name available to all modules we are required to make it into a **<u>global</u>** variable
- Global variables are available to all modules in the program
- In most programming languages, variables are declared explicitly as global or they are declared <u>within the main module</u> – outside of any modules

Functions - Scope

- Why not just use global variables?
- The advantages of local variables
  - Programmers can work on different modules without worrying about name duplication. Local variables with the same name in two different modules are stored in two different memory locations so statements affecting one **do not** affect the other
  - Local variables with the same name as a global variable are used over global

Functions Scope

- Global variables allow us to make variables available to other modules
  - Disadvantage is that if a module has to look up variables outside its own scope, this process takes extra time and requires programmers to be aware of all other variables being used in the program
  - An efficient way is to pass data as an argument
- Arguments are:
  - Values represented as constants or variables
  - Enclosed in parentheses following module name

Global variables in Python

- If you want to change the value of a global variable inside of a function, you must mark the variable with the keyword **global**
  - This is not required for list or dict containers since they are mutable

Module efficiency

- How do you know when to employ a module or include the code in your main module? When do you have a module return a value or display it?
- You consider **coupling** and **cohesion**

Module Efficiency

- Cohesion is the degree to which all statements and variables relate to one purpose
  - **Higher cohesion** is better, and it makes it so that the module can be used later for its single purpose
  - For example
    - A function that just adds two numbers and returns the result is much more cohesive than a function that adds two numbers **and** prints out the sum as a formatted statement.
  - The first example has a single purpose – adds two numbers
  - The second example has two purposes – add two numbers AND display the results

Module Efficiency

- Coupling is the degree to which a module depends on other modules to do its work
  - If the average function relied on global variables it would rely on the main module (would be highly coupled to it)
  - If a function contains multiple arguments, it also relies more heavily on the module calling it
  - **Lower coupling** is better

Module Efficiency

- Do NOT over modularize
  - Running modules does take computer power
  - Rule of thumb for creating a module
    - If you have more than five or six statements that are closely related to the same purpose (high cohesion) AND you have the potential of using the code in more than one place in your program (low coupling) – then consider turning those statements into a module
- Modularization advantages
  - Main module easier to understand
  - Can divide work among programmers
  - Modules can be reused in a program

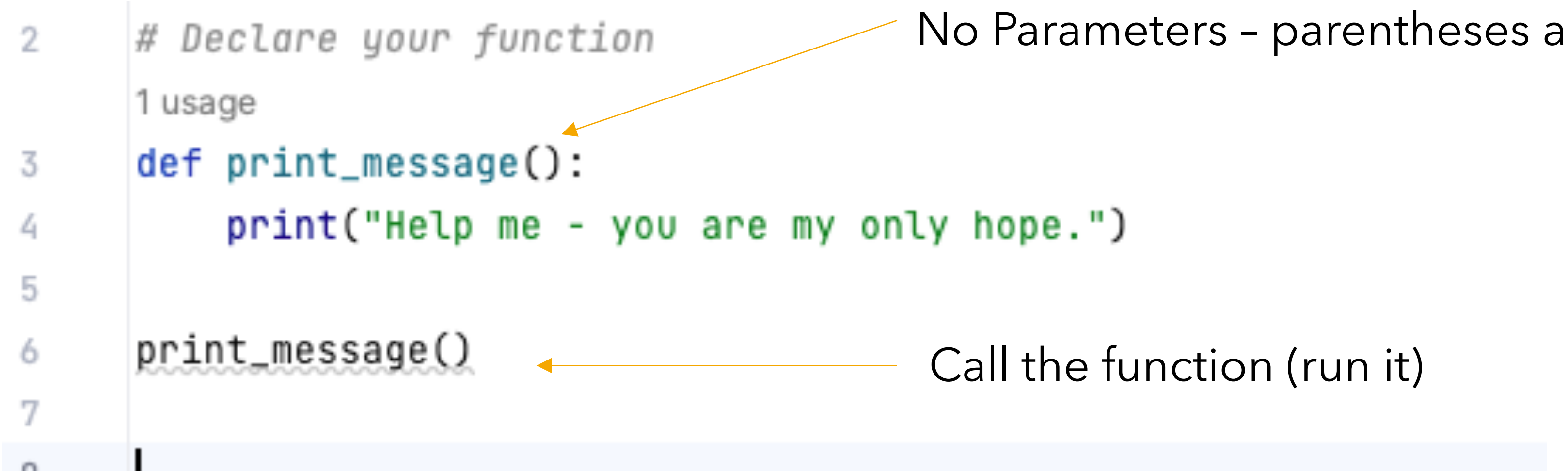
Basic functions

- A function is defined with the keyword `def`
- Parentheses wrap **arguments** – which are optional
- A : marks the end of the function name
- Indentation indicates the body of the function

Basic functions

- Once we have created the function we want – we can use it by **calling** the function
- With a basic function, we simply use the name that we have given the function and place parentheses after it ( )

Basic Functions



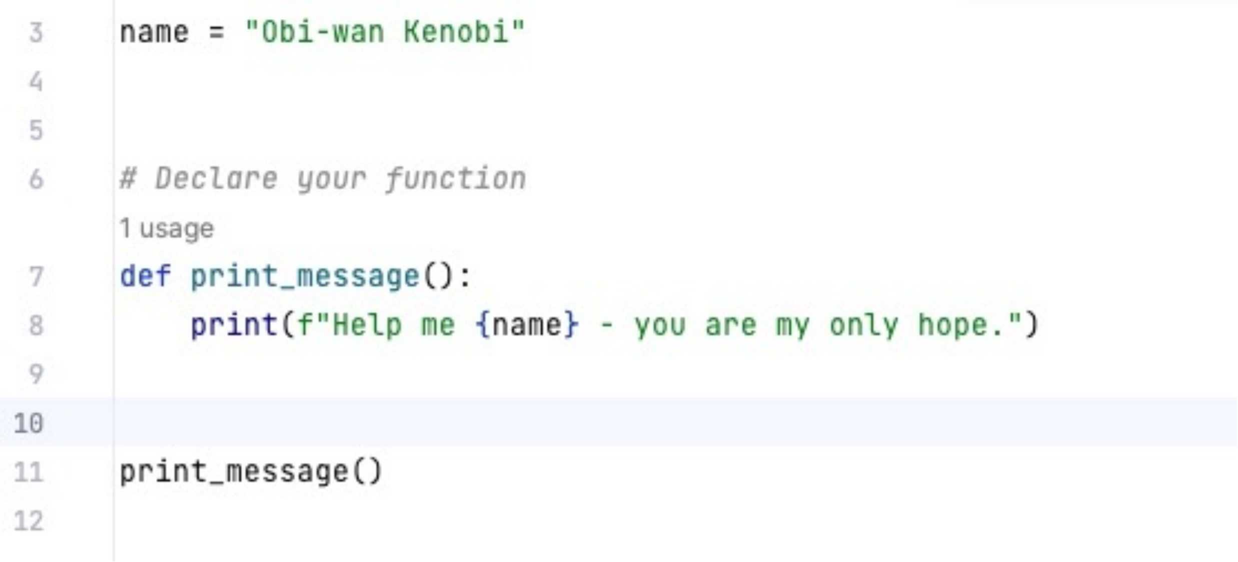
```
# Declare your function
1 usage
def print_message():
    print("Help me - you are my only hope.")

print_message()

```

No Parameters – parentheses are still required

Call the function (run it)

Arguments

- Functions may require additional information in order for them to work properly.
- For example – a function that prints out a name. This requires you to provide it with a name to print out.
    - Option 1 is to use a global variable in order to accomplish this:

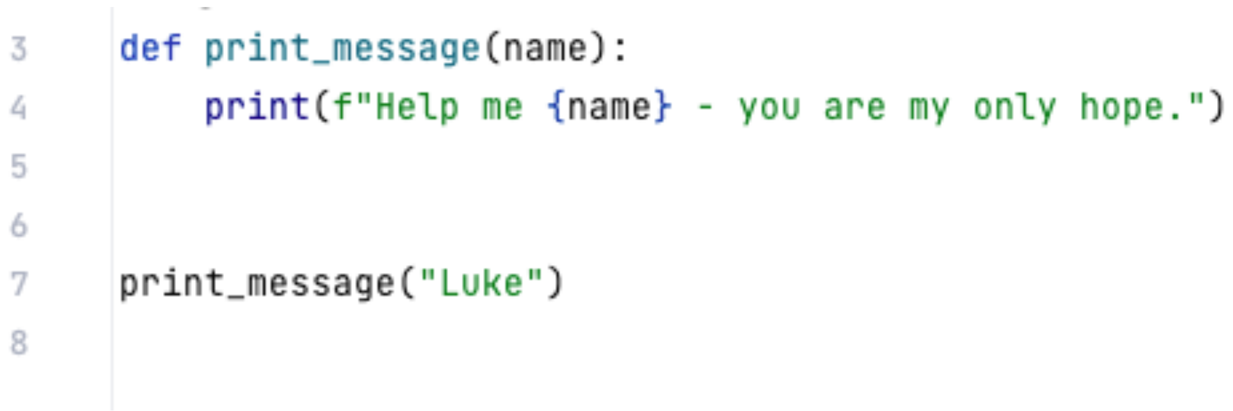


```
name = "Obi-wan Kenobi"


# Declare your function
1 usage
def print_message():
    print(f"Help me {name} - you are my only hope.")


print_message()

```

Any program that we use it with, we MUST create a variable called `name` – this function is **highly coupled**

Arguments

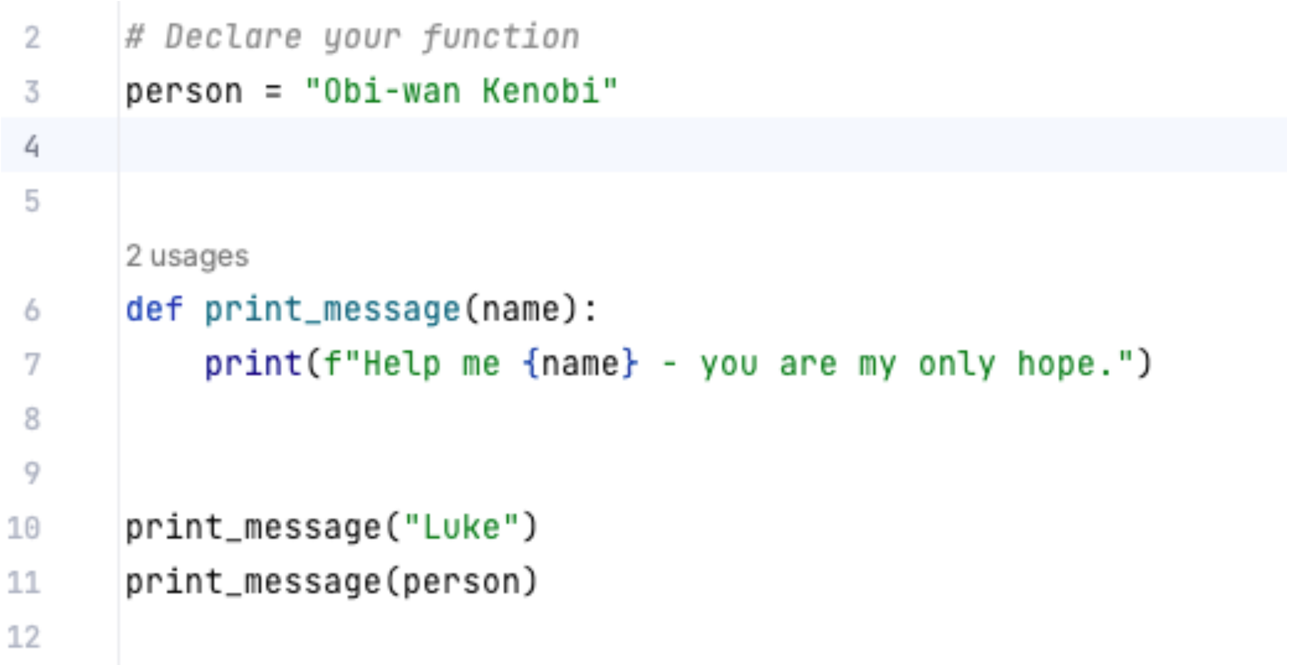
- Option 2 – pass an **argument** (or value) to the function. This ensures that as long as the value is the type we want, we can execute our code
- This value is represented in our code with a **parameter** name



```python
def print_message(name):
    print(f"Help me {name} - you are my only hope.")


print_message("Luke")

```

Parameters

- Parameters allow us to reduce our **coupling**
- They allow the outcomes to be different based on the different values that get passed



```python
# Declare your function
person = "Obi-wan Kenobi"


def print_message(name):
    print(f"Help me {name} - you are my only hope.")


print_message("Luke")
print_message(person)

```

Functions – passing multiple arguments

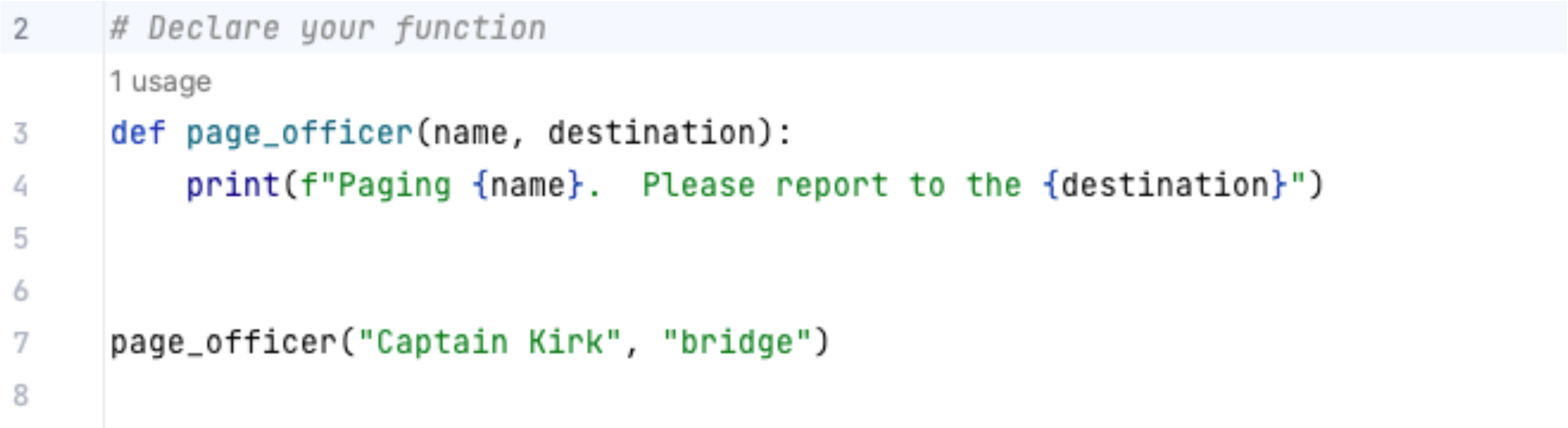
- Modules can be designed to accept more than one argument
- Parameters must match the arguments that are sent, and they must match in the number AND the order
  - If the wrong number of arguments are sent, then a **syntax error** will occur
  - If the order is mixed up, then a **logic error** occurs

Functions - passing multiple arguments

- Arguments may be different data types
- Limit number of arguments to five or six to improve a program`s readability
    - If more, break up module into two

Function Parameters

- A function can take as many parameters as required to perform its task
- The parameters are all defined with a type and separated with a ,



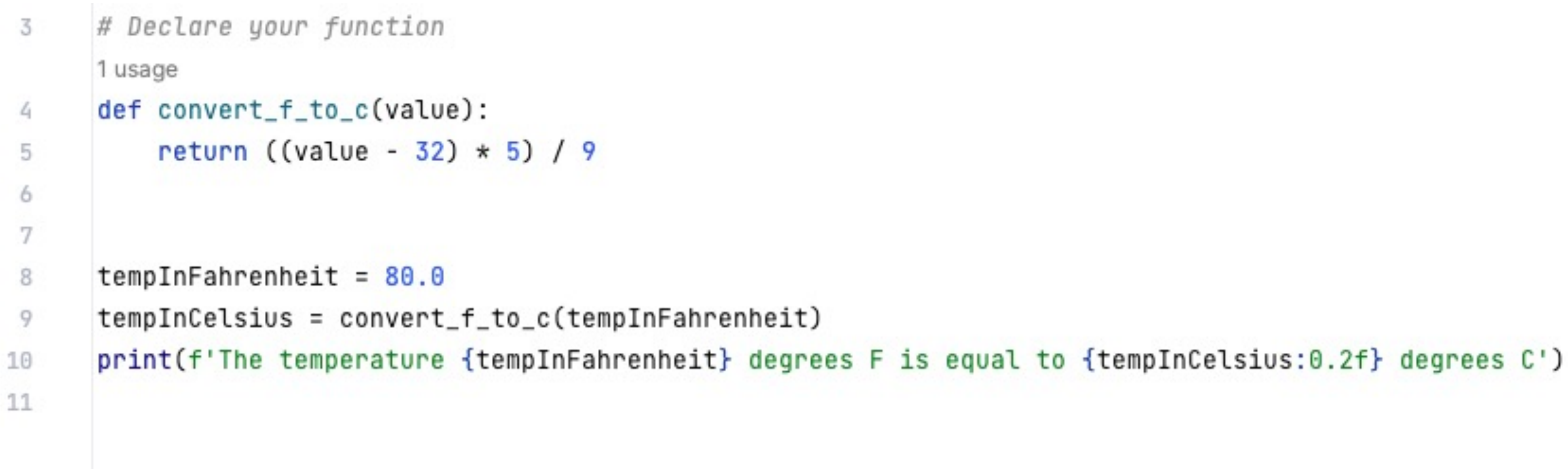
```
# Declare your function
1 usage
def page_officer(name, destination):
    print(f"Paging {name}.  Please report to the {destination}")


page_officer("Captain Kirk", "bridge")

```

Functions – Returning Values

- Modules can be used and performed without the main module knowing anything about the results
  - Modules that display information for example
- Sometimes data is needed – these modules can send a ***return value*** back to the calling module
  - Although multiple arguments can be sent to the modules and functions , only *one value can be returned*

Returning from a function

- To specify a return value you provide the **return** keyword as part of your function definition
  - Capture the returned value by assigning it to a variable



```
# Declare your function
1 usage
def convert_f_to_c(value):
    return ((value - 32) * 5) / 9


tempInFahrenheit = 80.0
tempInCelsius = convert_f_to_c(tempInFahrenheit)
print(f'The temperature {tempInFahrenheit} degrees F is equal to {tempInCelsius:0.2f} degrees C')

```

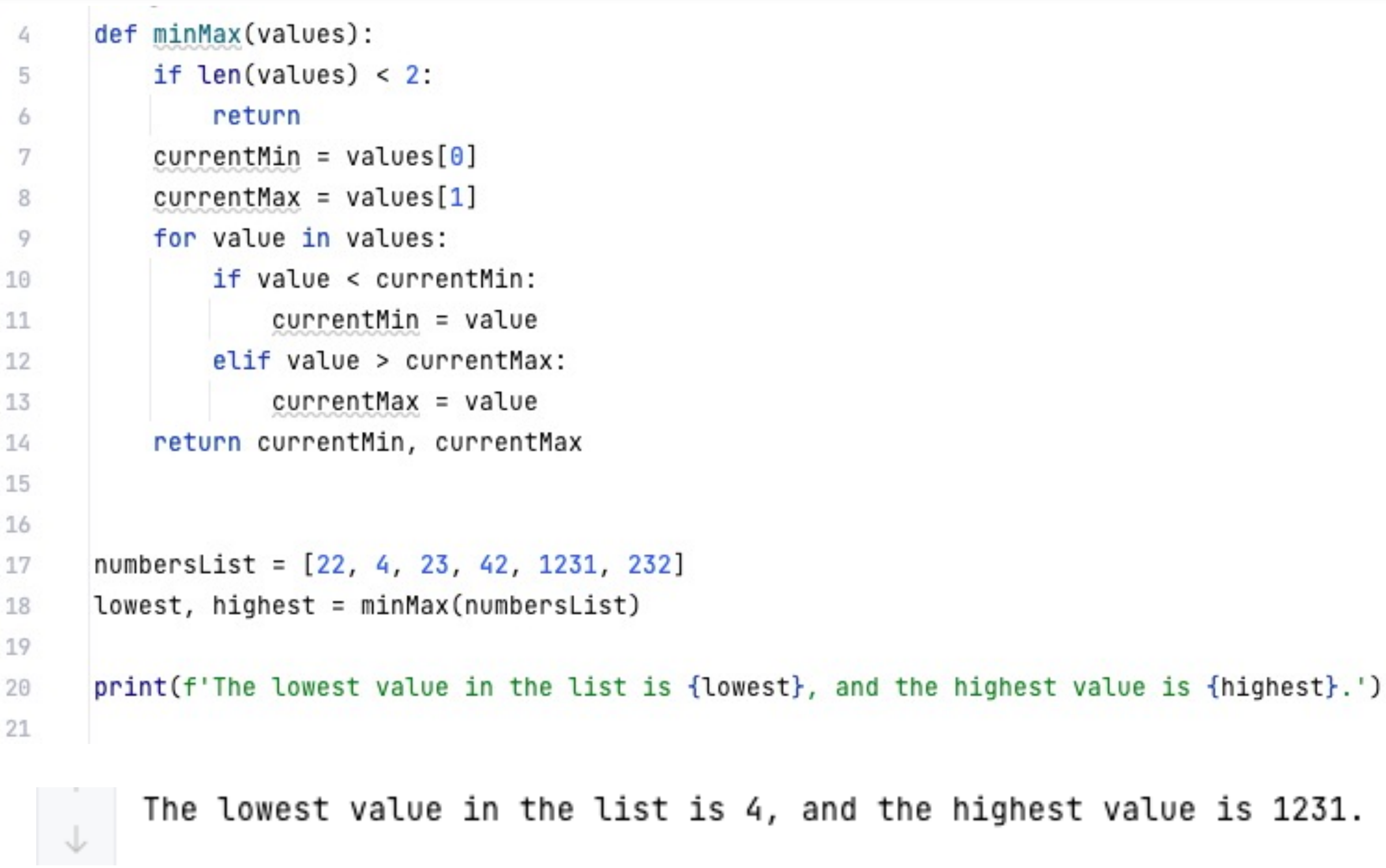
Functions – Returning Values

- Functions with no return statement are a **void** function
- These functions return a value of **None**
  - None indicated no value

Multiple Returns

- Functions return a single value
- Functions that return a tuple type as the return type are able to return multiple values as a single compound return value
  - You can unpack the values of the tuple, by assignment them to multiple variables

Multiple Return Types



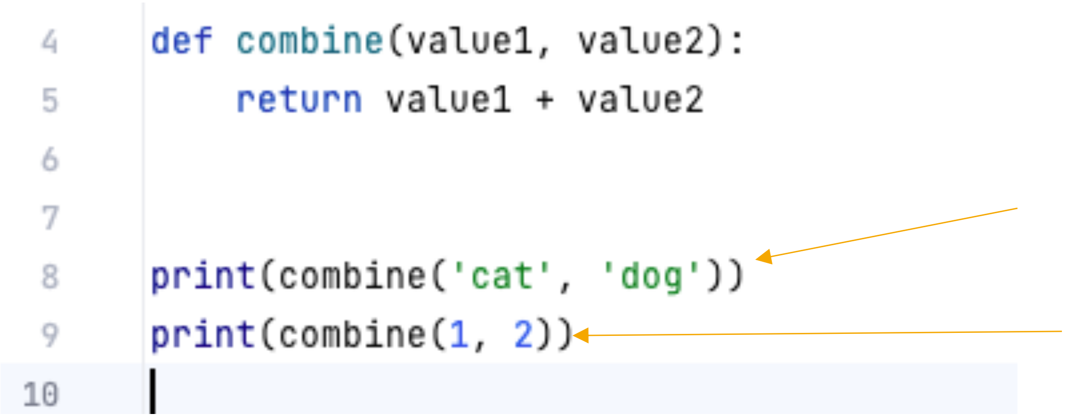
```
def minMax(values):
    if len(values) < 2:
        return
    currentMin = values[0]
    currentMax = values[1]
    for value in values:
        if value < currentMin:
            currentMin = value
        elif value > currentMax:
            currentMax = value
    return currentMin, currentMax


numbersList = [22, 4, 23, 42, 1231, 232]
lowest, highest = minMax(numbersList)

print(f'The lowest value in the list is {lowest}, and the highest value is {highest}.')
```

```
The lowest value in the list is 4, and the highest value is 1231.
```

Parameter types

- Functions can take any type of object as arguments
- This ability to work with different types easily is known as **polymorphism**

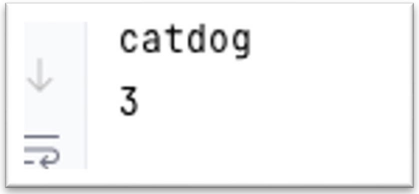
Parameter Types



```python
def combine(value1, value2):
    return value1 + value2


print(combine('cat', 'dog'))
print(combine(1, 2))
```

Works with strings

Works with numbers



```
catdog
3
```

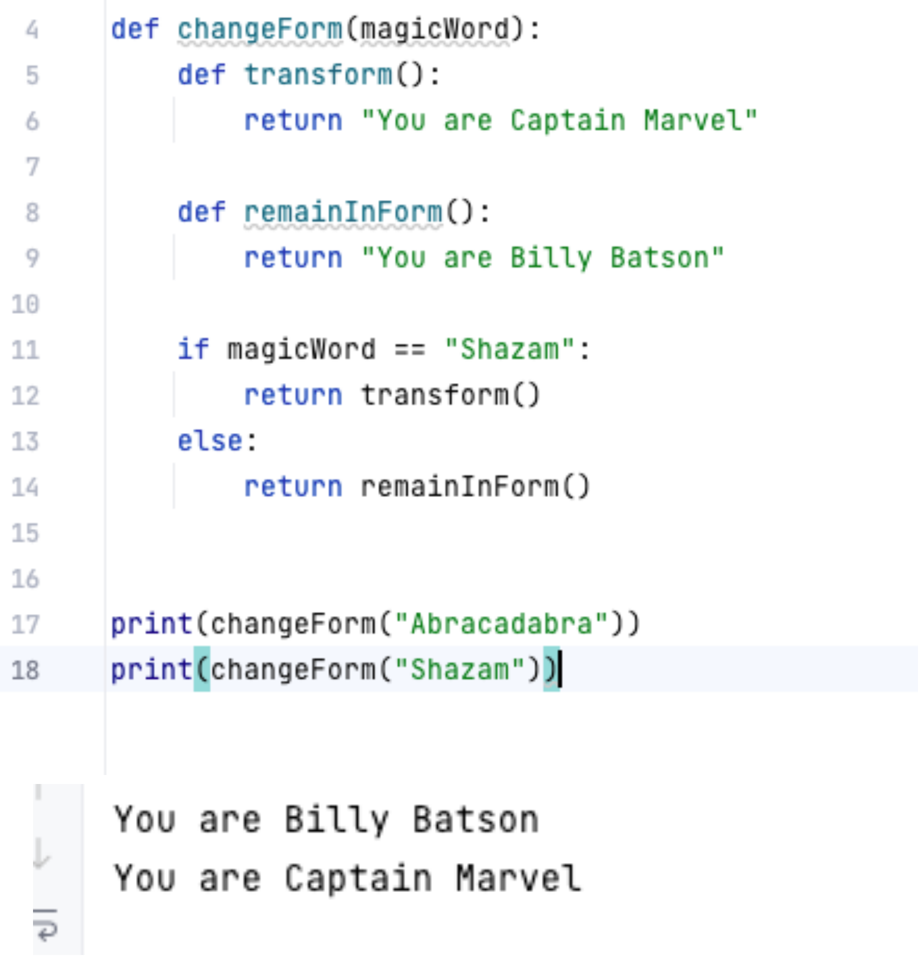
Dynamic vs. Static

- Dynamic typing determines the type of objects during execution
  - Interpreter is responsible for checking operations as the program executes
- Static typing defines the types of objects during declaration
  - Compiler performs checks and halts compilation when errors are found

Nested Functions & Scope

- Functions can be defined within the bodies of other functions (nested functions
  - Nested functions are hidden from code outside the function it is nested in
  - An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope
  - Useful when you want to hide or isolate parts of your code

Nested Functions



```python
def changeForm(magicWord):
    def transform():
        return "You are Captain Marvel"

    def remainInForm():
        return "You are Billy Batson"

    if magicWord == "Shazam":
        return transform()
    else:
        return remainInForm()


print(changeForm("Abracadabra"))
print(changeForm("Shazam"))
```

```
You are Billy Batson
You are Captain Marvel
```

Incremental Development

- Programs are written using incremental development
  - Write a small amount of code
  - Test the code
  - Write more code
  - Test the code
  - …

Function Stubs

- Function stubs are ways to implement incremental development
  - Allow to create basic functional components
  - The pass keyword can be used as a placeholder for content
    - It allows the program to ‘function’ until the contents have been fully implemented

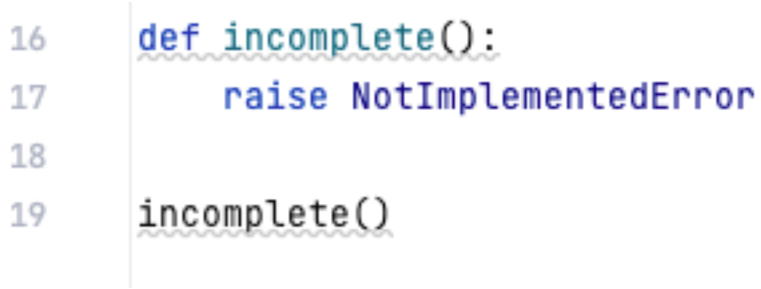
Function Stubs

- Cases can also make use of print statements or invalid return numbers

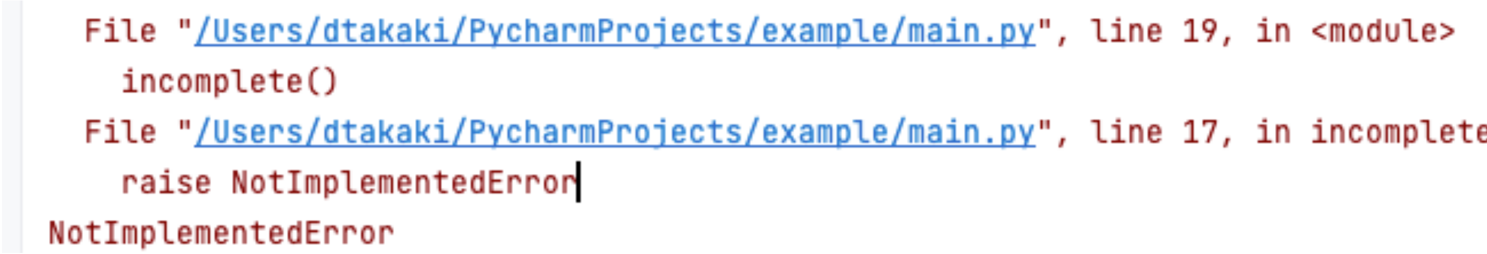
```
print('FIXME: Complete implementation of functionNameGoesHere')
```

Function Stubs

- You may want a program to stop when incomplete code is reached
- Implement a **NotImplementedError**



```
16  def incomplete():
17      raise NotImplementedError
18
19  incomplete()
```

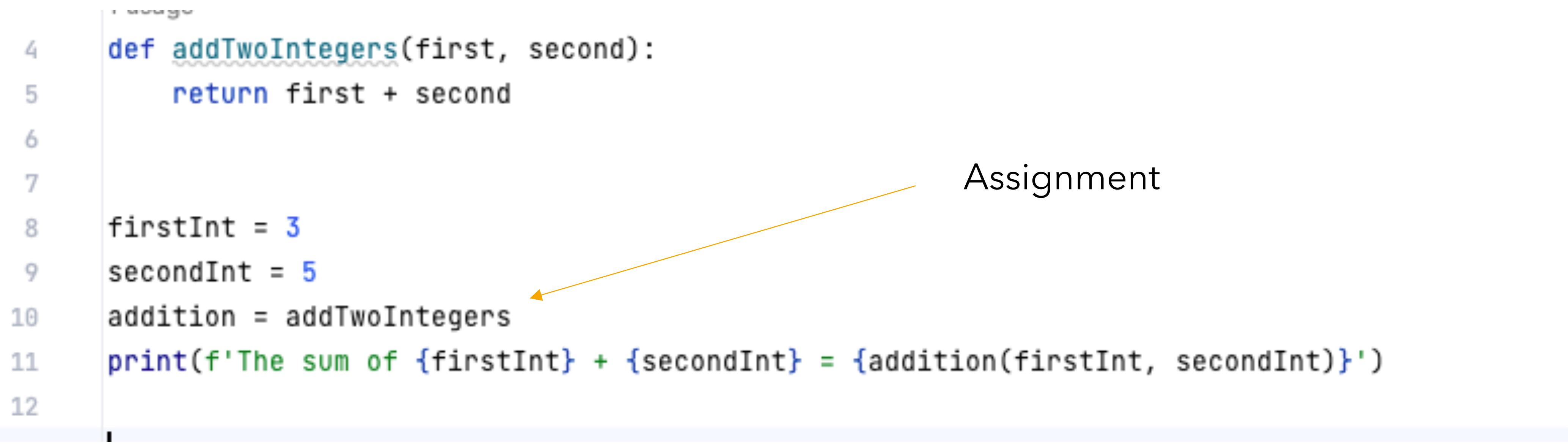


```
  File "/Users/dtakaki/PycharmProjects/example/main.py", line 19, in <module>
    incomplete()
  File "/Users/dtakaki/PycharmProjects/example/main.py", line 17, in incomplete
    raise NotImplementedError
NotImplementedError
```

Functions are Objects

- A function is an object
  - They have a type, identity and value
  - A function creates a new function object with its name bound to that object
- This allows functions to be assigned and used as parameters

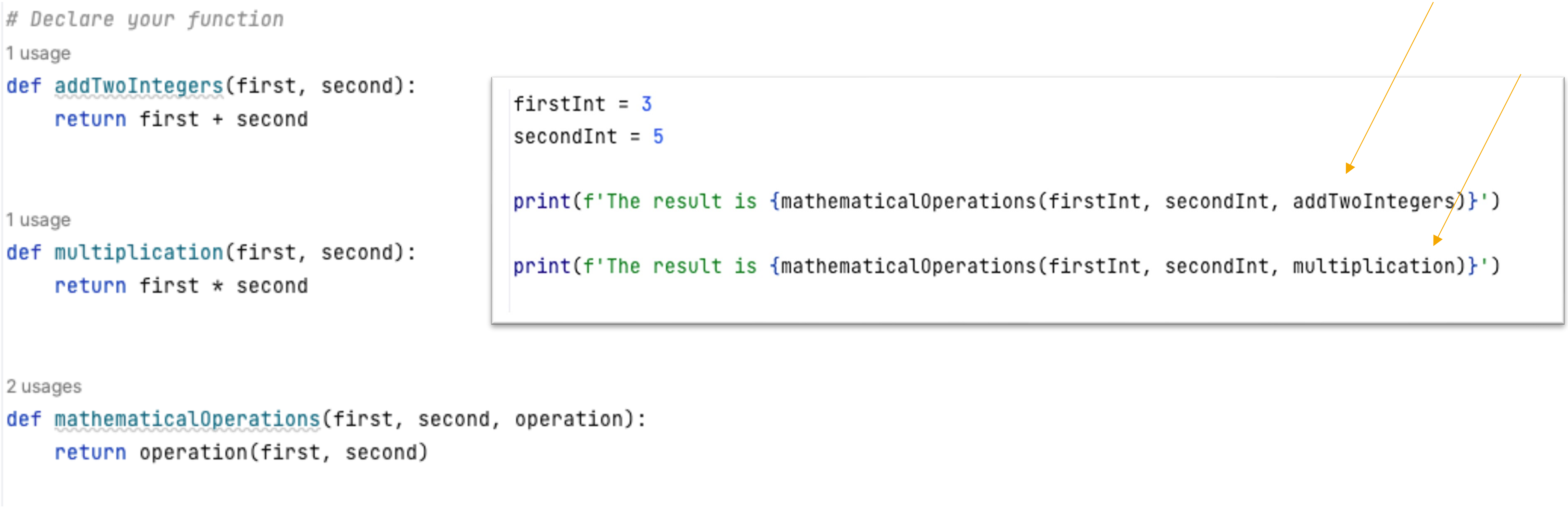
Functions as objects - assignment



```
1 usage
def addTwoIntegers(first, second):
    return first + second


firstInt = 3
secondInt = 5
addition = addTwoIntegers
print(f'The sum of {firstInt} + {secondInt} = {addition(firstInt, secondInt)}')

```

Assignment

Functions as objects - arguments



```
# Declare your function
1 usage
def addTwoIntegers(first, second):
    return first + second


1 usage
def multiplication(first, second):
    return first * second


2 usages
def mathematicalOperations(first, second, operation):
    return operation(first, second)
```

```
firstInt = 3
secondInt = 5

print(f'The result is {mathematicalOperations(firstInt, secondInt, addTwoIntegers)}')

print(f'The result is {mathematicalOperations(firstInt, secondInt, multiplication)}')
```

Namespace

- A namespace maps names to objects
  - This is a dictionary whose keys are the names and values are the objects
  - Local namespace values can be found using **locals()** function
  - Global namespace values can be found using **global()** function

Keyword arguments

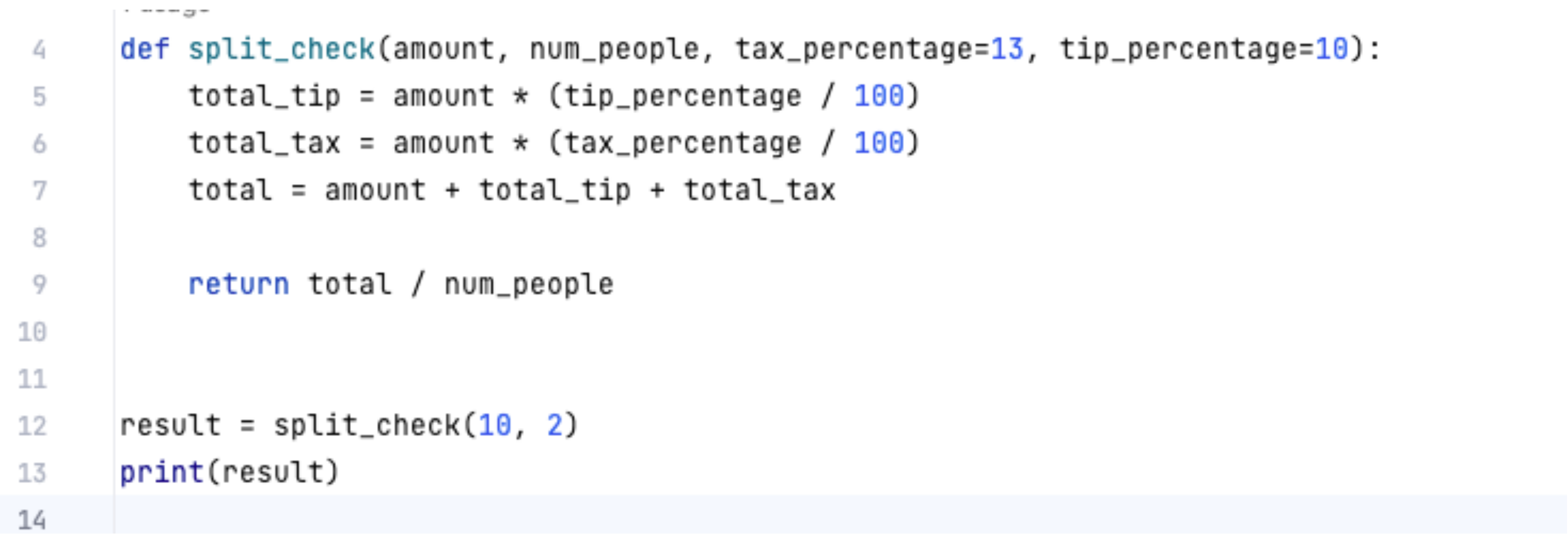
- Functions may require multiple arguments
  - This makes the contents difficult to read and can allow for the user to make an error when placing the arguments in the correct order
- Keyword arguments allow for the mapping of arguments to parameters with specific names – not relying on their position
  - Order is no longer important

Keyword arguments

- Good practice is to use keyword arguments for four or more arguments
    - They can be mixed with positional elements

Default Parameter Values

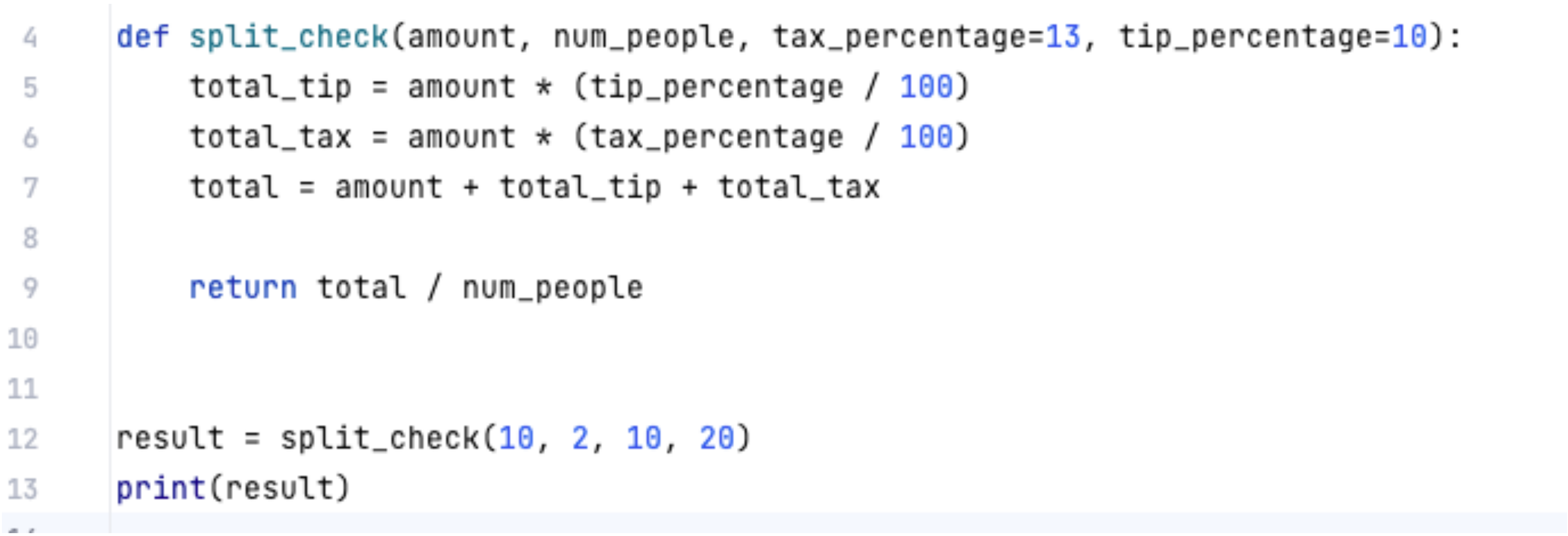
- Parameters in a function can take default values
  - Default values are written using the assignment operator
  - Call the function and pass no values to use the default value
  - Call the function and pass a value to override the default value



```python
def split_check(amount, num_people, tax_percentage=13, tip_percentage=10):
    total_tip = amount * (tip_percentage / 100)
    total_tax = amount * (tax_percentage / 100)
    total = amount + total_tip + total_tax

    return total / num_people


result = split_check(10, 2)
print(result)

```

Default Parameter Values

- The default parameter values can be overridden

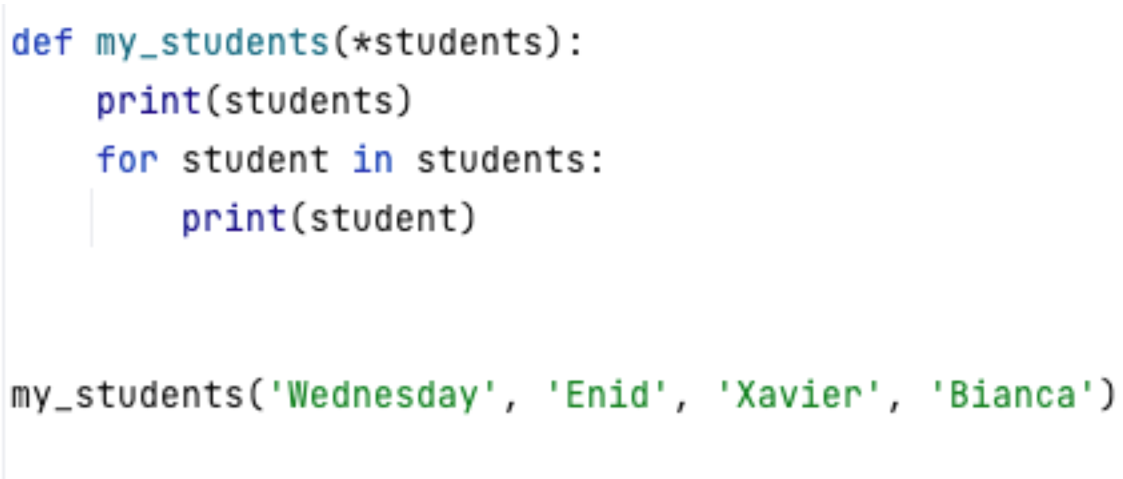


```python
def split_check(amount, num_people, tax_percentage=13, tip_percentage=10):
    total_tip = amount * (tip_percentage / 100)
    total_tax = amount * (tax_percentage / 100)
    total = amount + total_tip + total_tax

    return total / num_people


result = split_check(10, 2, 10, 20)
print(result)
```

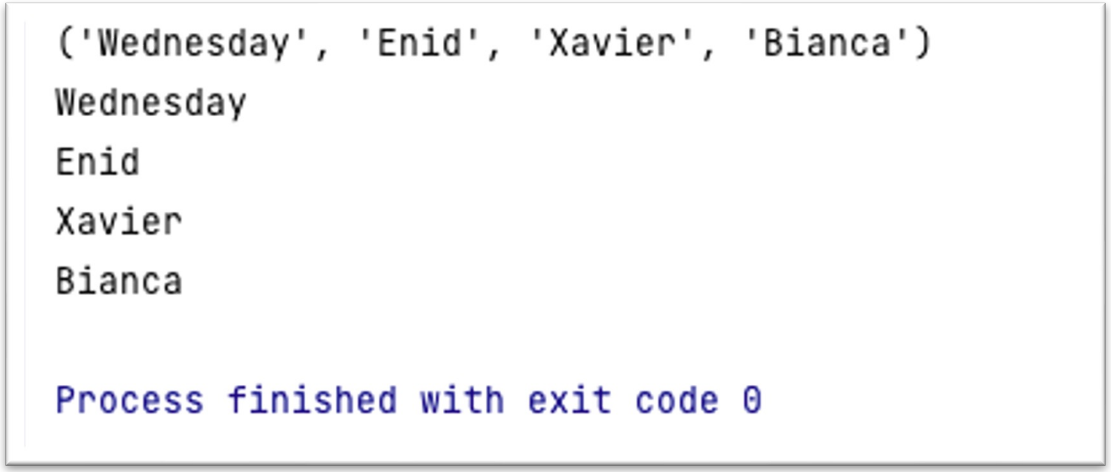
Arbitrary Arguments

- When a function can take one or more values, they can be collected using an arbitrary argument list
- Identified with the *args parameter
- This is a tuple of values

Arbitrary Arguments



```python
def my_students(*students):
    print(students)
    for student in students:
        print(student)


my_students('Wednesday', 'Enid', 'Xavier', 'Bianca')
```

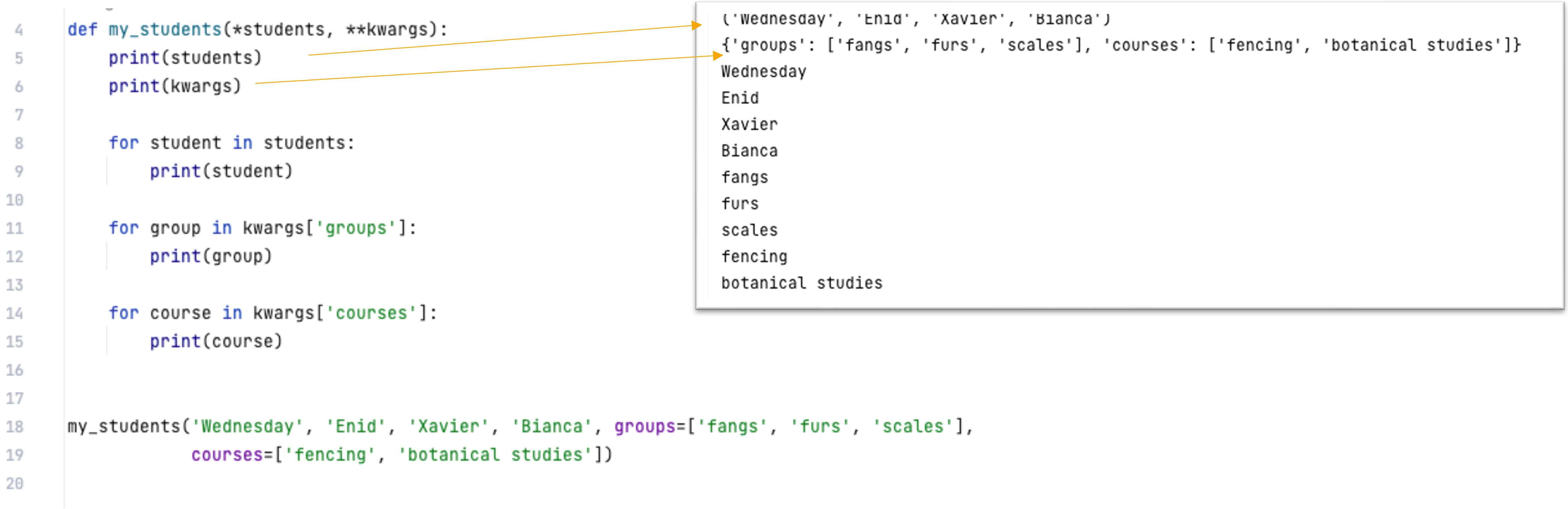


```
('Wednesday', 'Enid', 'Xavier', 'Bianca')
Wednesday
Enid
Xavier
Bianca

Process finished with exit code 0
```

Keyword Arguments

- Multiple keyword arguments can be added use **kwargs parameter
- This creates a dictionary of extra arguments
  - Keys are the parameter names in the function call

Keyword Arguments



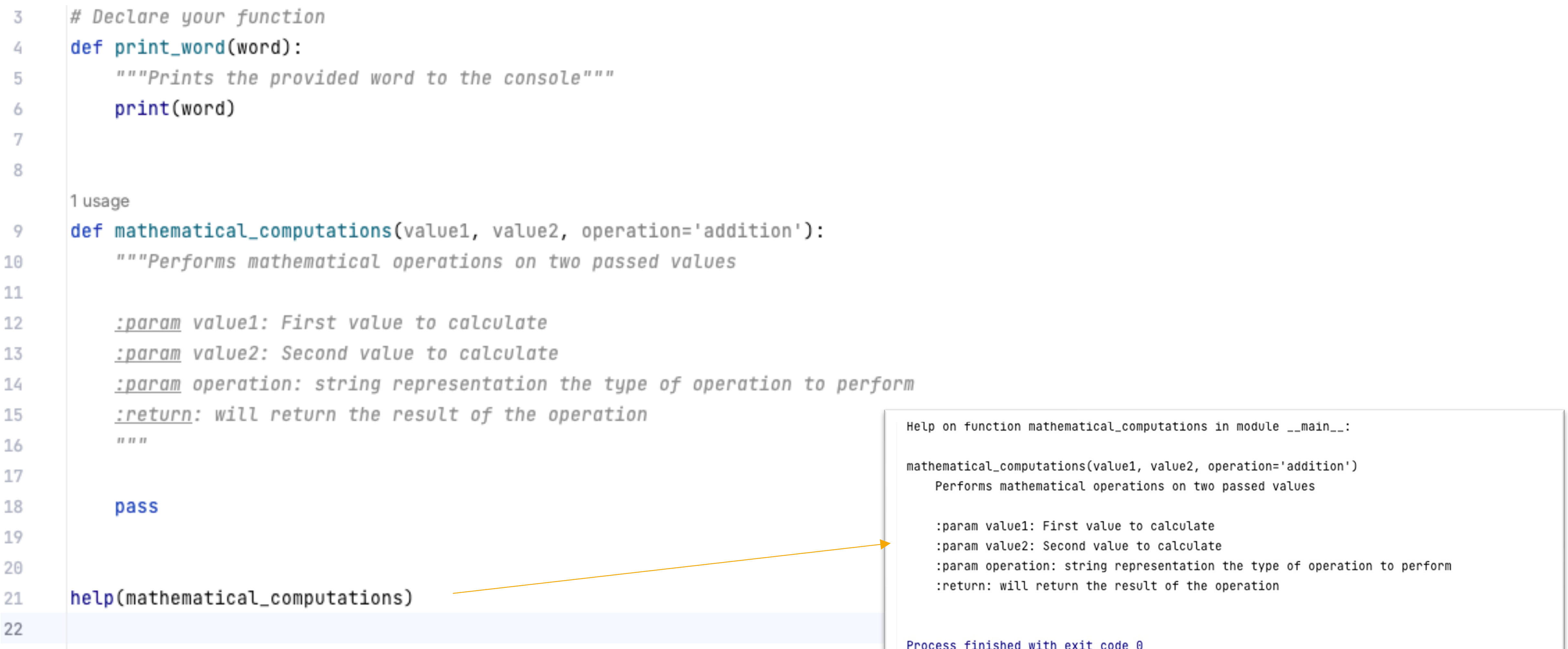
```python
def my_students(*students, **kwargs):
    print(students)
    print(kwargs)

    for student in students:
        print(student)

    for group in kwargs['groups']:
        print(group)

    for course in kwargs['courses']:
        print(course)


my_students('Wednesday', 'Enid', 'Xavier', 'Bianca', groups=['fangs', 'furs', 'scales'],
            courses=['fencing', 'botanical studies'])
```

```
('Wednesday', 'Enid', 'Xavier', 'Bianca')
{'groups': ['fangs', 'furs', 'scales'], 'courses': ['fencing', 'botanical studies']}
Wednesday
Enid
Xavier
Bianca
fangs
furs
scales
fencing
botanical studies
```

Docstrings

- Documenting your code is important
  - It helps you understand your own code
  - It helps others understand your code
- A docstring is a literal value placed in the first line of a function body
- A docstring starts and ends with “””

Docstrings



```python
# Declare your function
def print_word(word):
    """Prints the provided word to the console"""
    print(word)


1 usage
def mathematical_computations(value1, value2, operation='addition'):
    """Performs mathematical operations on two passed values

    :param value1: First value to calculate
    :param value2: Second value to calculate
    :param operation: string representation the type of operation to perform
    :return: will return the result of the operation
    """

    pass


help(mathematical_computations)
```

```
Help on function mathematical_computations in module __main__:

mathematical_computations(value1, value2, operation='addition')
    Performs mathematical operations on two passed values

    :param value1: First value to calculate
    :param value2: Second value to calculate
    :param operation: string representation the type of operation to perform
    :return: will return the result of the operation


Process finished with exit code 0
```