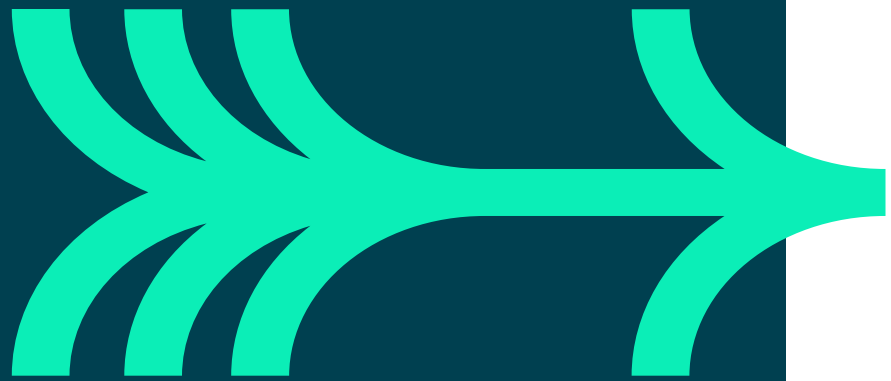
Functions and data



The Topic: What?

- An introduction to code reuse in Python
  - Functions
  - Parameters
  - Variadic Functions
  - Scope
  - Docstrings

Applications: Why?

- To be able to make code sharable and re-useable

Expectations: Who?

- Learners are expected to have covered fundamental programming in Python previously.

Functions

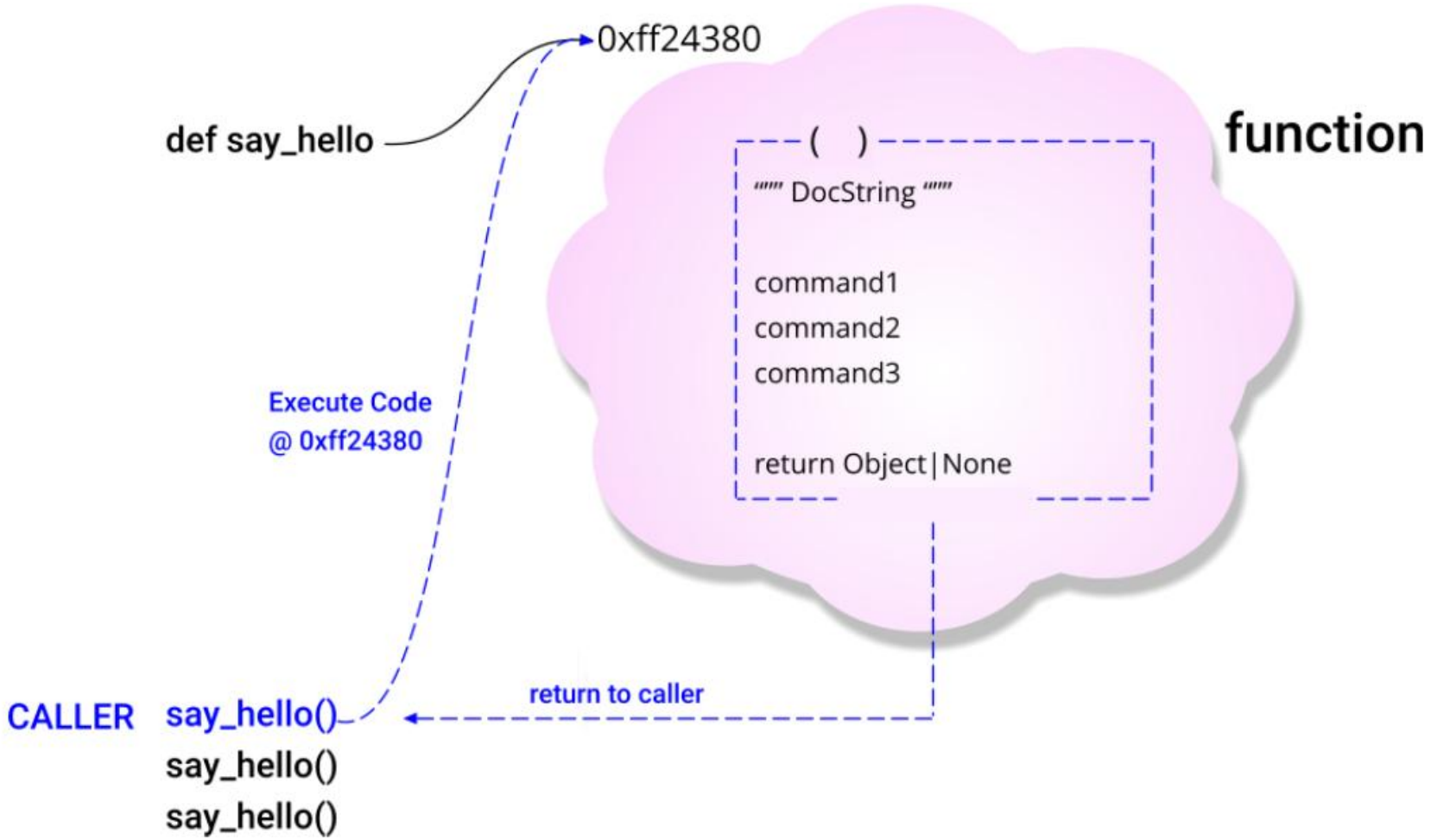
The Art of reusing named blocks of code

Functions

Earlier in the DIGITAL session, you learned about functions and how useful they are for making code sharable and reusable. You saw function definitions, passing parameters to a function, returning a value from a function, and documenting functions with docstrings.

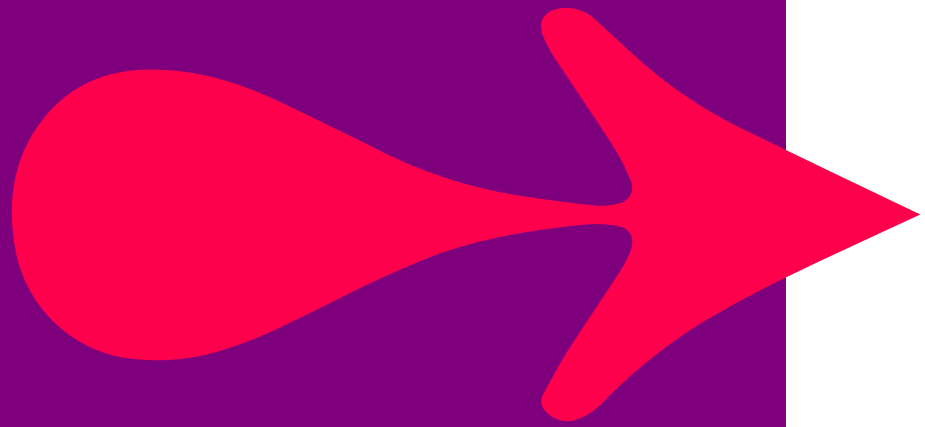
You can do more with functions! You will learn about different types of parameters, variable scope, the idea of annotating functions, lambda functions, and using functions as objects. This includes passing a function as a parameter and returning a function from another function.

Functions


0xff24380
def say_hello
function
( )
""" DocString """
command1
command2
command3
return Object|None
Execute Code
@ 0xff24380
return to caller
CALLER say_hello()
say_hello()
say_hello()

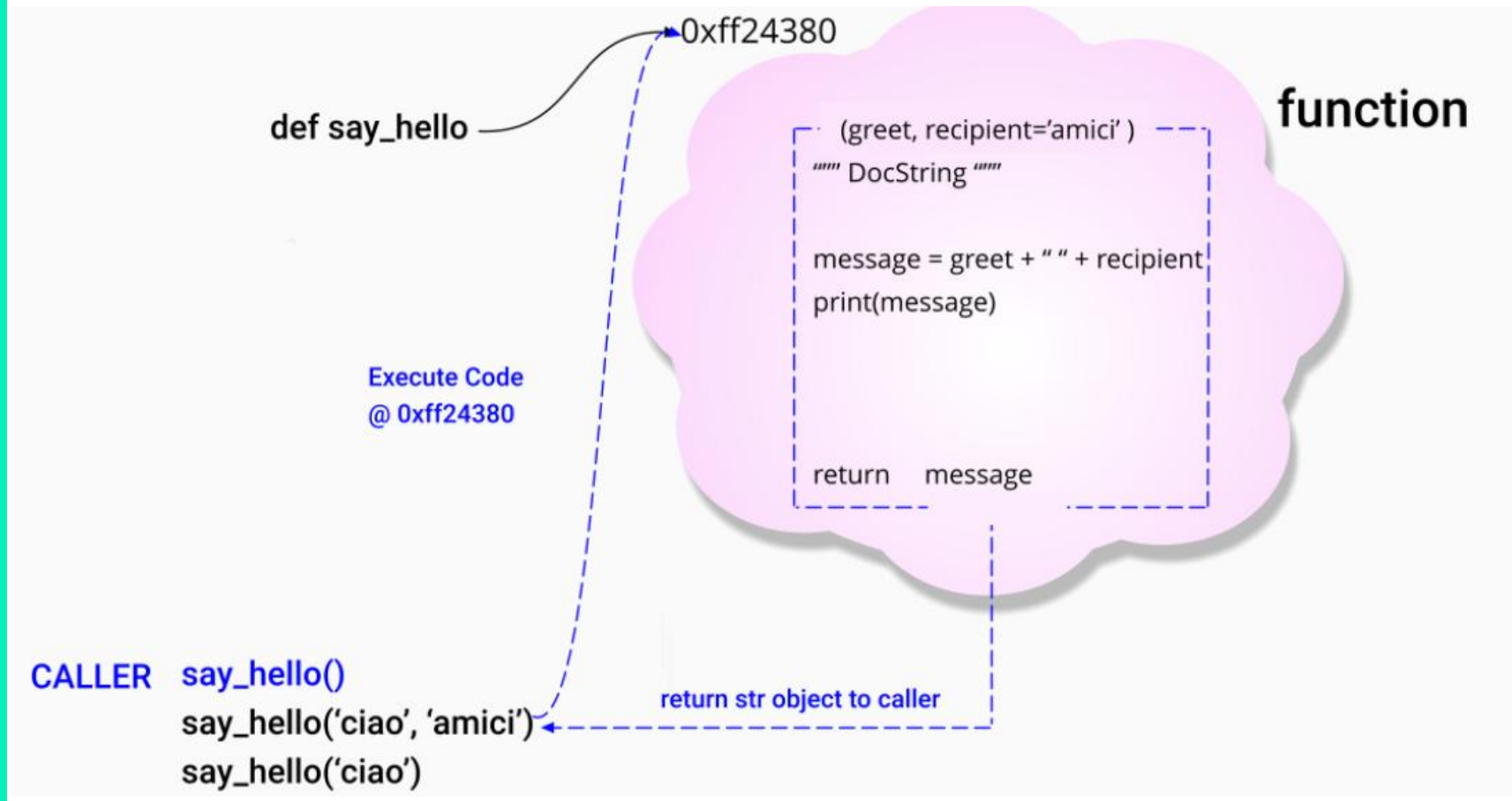
Functions

Trainer demonstration



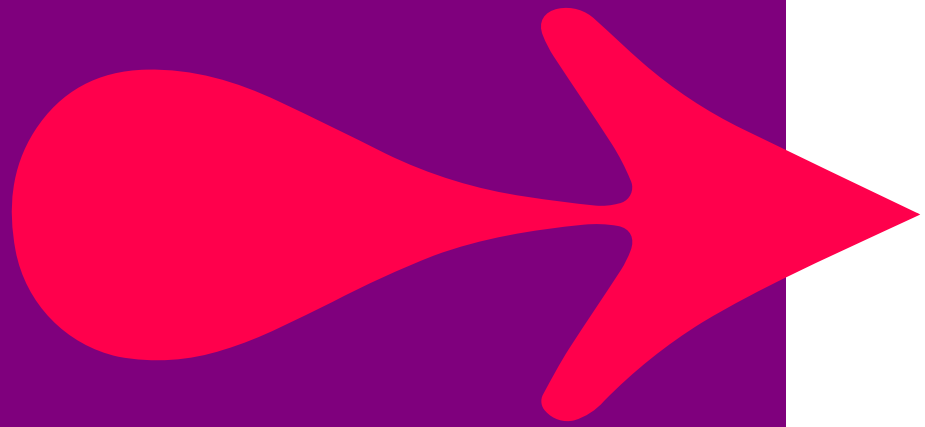
demo_user_functions.py

Parameters


0xff24380
def say_hello
function
(greet, recipient='amici')
""" DocString """
message = greet + " " + recipient
print(message)
return message
Execute Code
@ 0xff24380
CALLER say_hello()
say_hello('ciao', 'amici')
say_hello('ciao')
return str object to caller

Parameters

Trainer demonstration



demo_user_functions_with_parameters.py

User functions can optionally accept a finite number of parameters.

· Formal parameter names are defined within the parentheses of function declaration

· Actual values are passed in the CALLER's parentheses

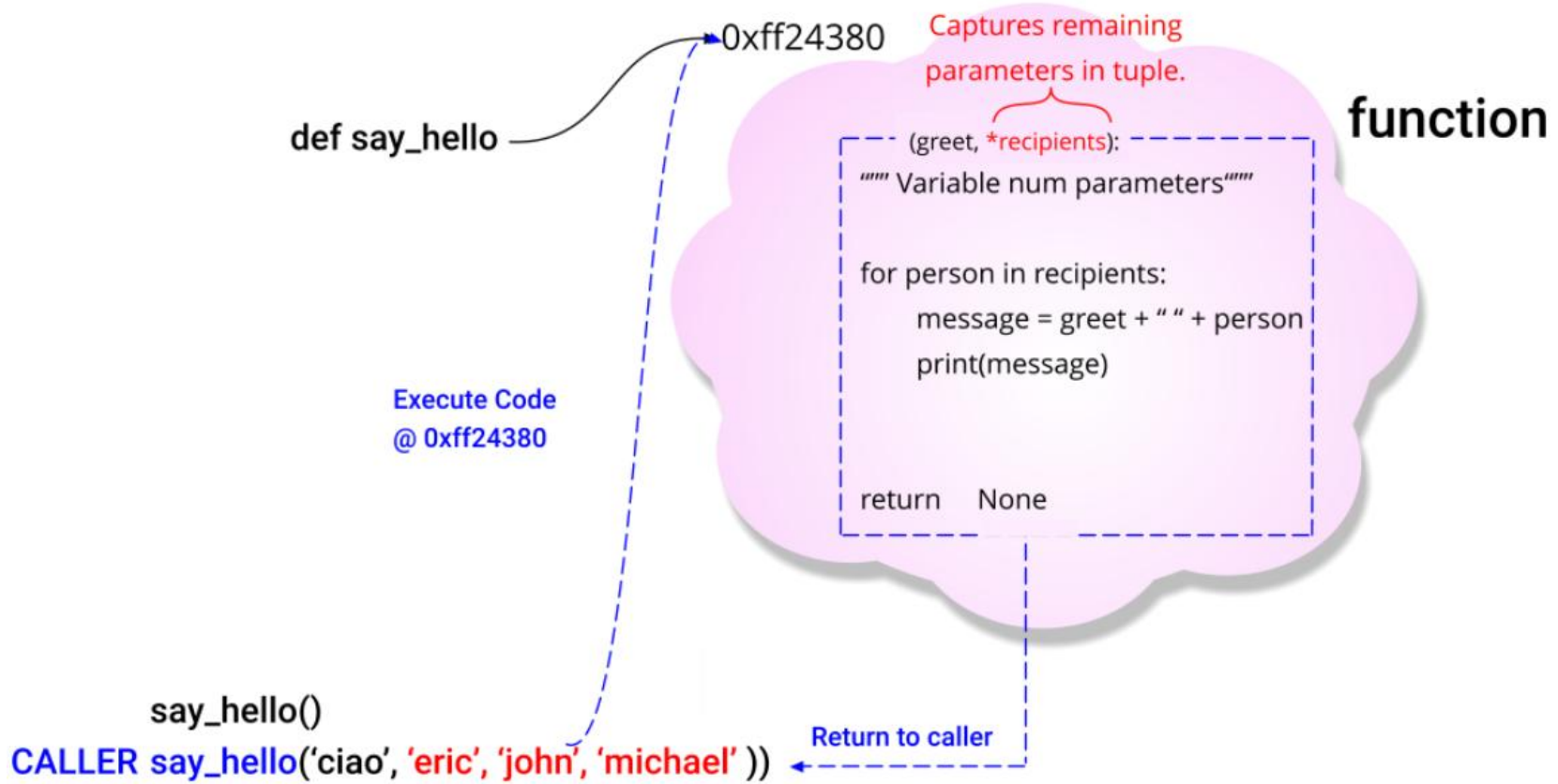
· Literal values are passed by COPY/VALUE, e.g. (100, 200, 'hello')

· Named objects are passed by Reference, e.g. (pattern, filename). So changing a parameter within a function will alter the caller's object. To prevent this for lists and dictionaries, pass a slice of the structure. e.g. function_call(my_list[:]). It's a hack!

· Parameters can be assigned defaults, but following parameters must also have defaults

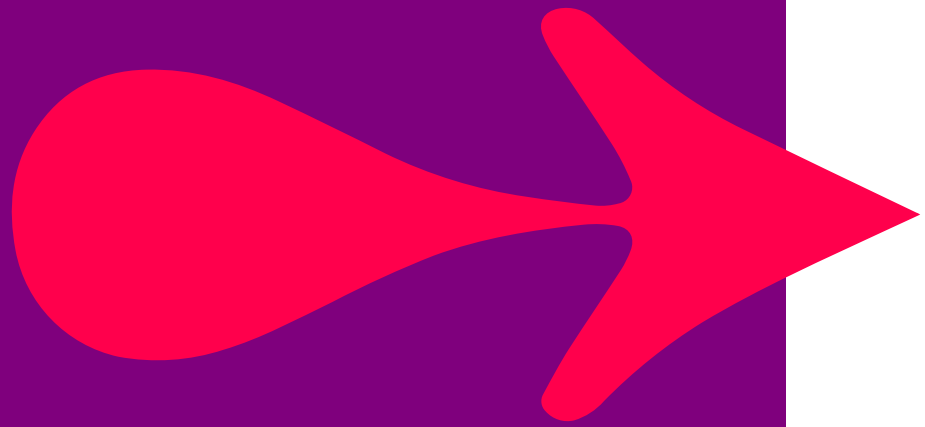
· Parameters can be passed as positional, named or a mix (positional followed by named)

Variadic functions



Variadic functions

Trainer demonstration



demo_user_functions_variadic.py

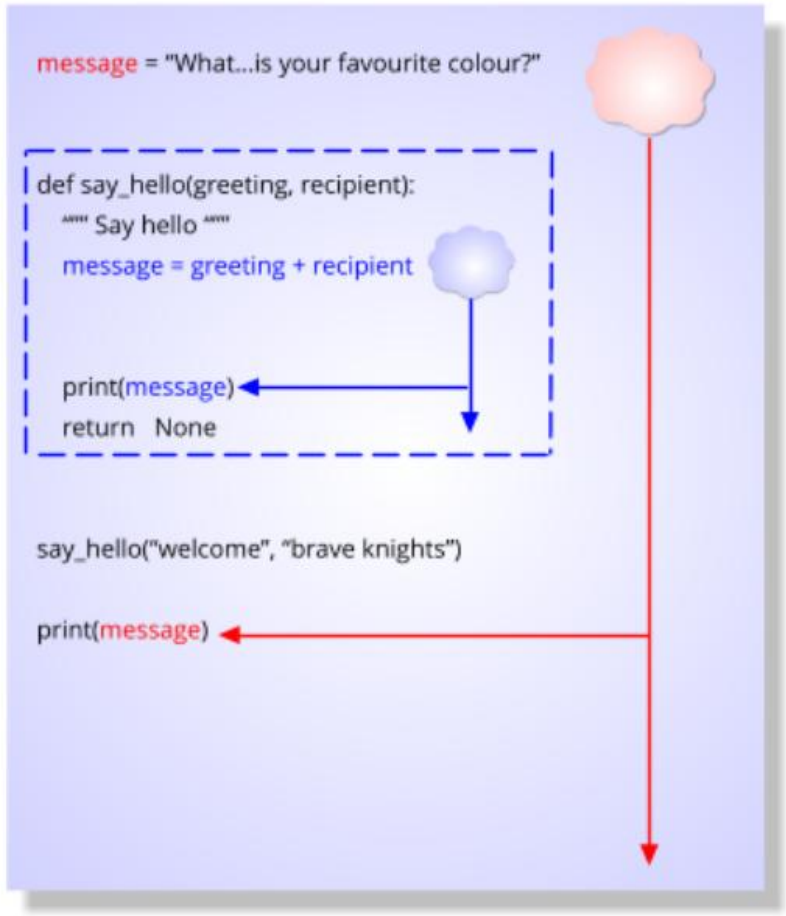
A **Variadic function** can accept a variable number of parameters. For example, the built-in function print() can accept any number of parameters. User functions can also accept variable number of parameters into either a **tuple** with a **'*'** prefix or into a **dictionary** with a **'**'** prefix. This works in a similar way to unpacking as discussed with collections.

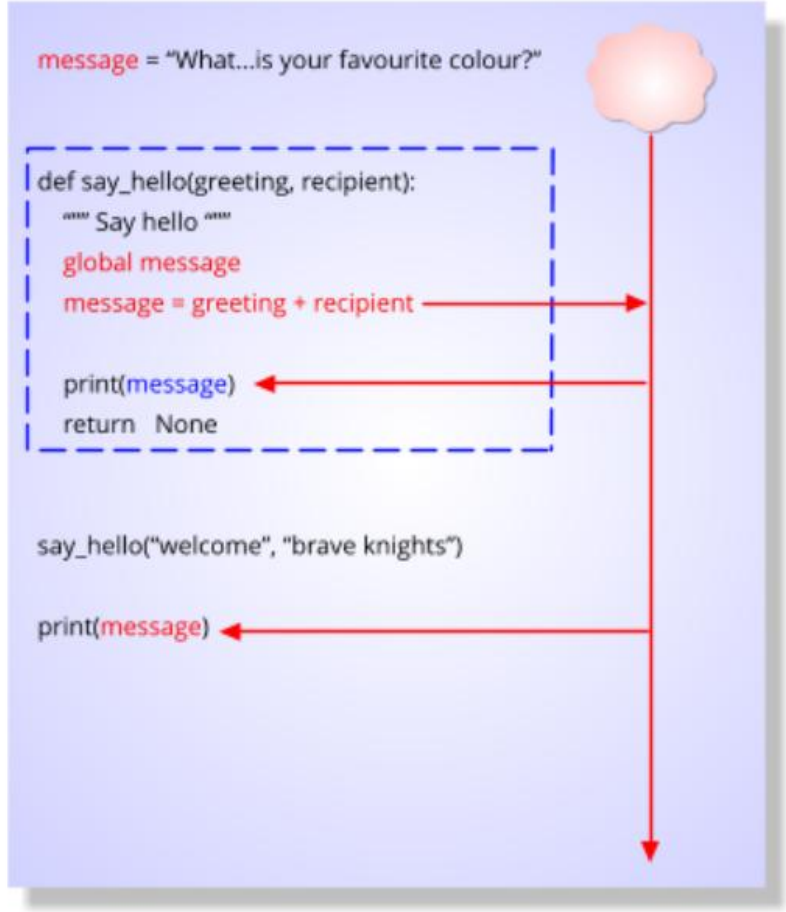
DEFINITION: def display_vat(**kwargs)

CALLER: display_vat(vatpc=15, gross=9.55, message='Summary') # Use named parameters when passing to a dict.

Scope

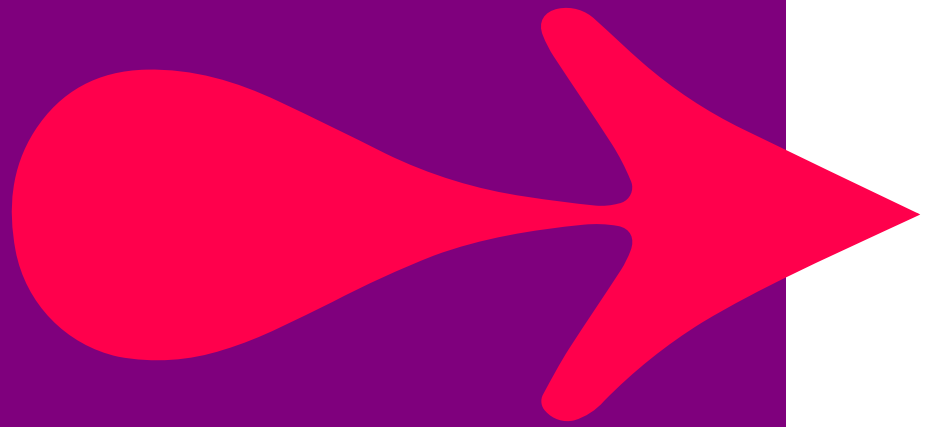
Local Vs Global





Scope

Trainer demonstration



demo_user_functions_scope.py

Scope defines the **life** and **visibility** of a variable/object, in other words, what part of the code can see and use it. Python supports global scope and function/local scope. Global variables/objects are visible to the entire program including nested functions.

But if a value is assigned in the function before it is used, then it becomes a local variable (visible and lives for duration of function).

To change the value of an outer global variable within a function (frowned upon), declare the variable as **global**.

Docstrings

So have you been wondering why we have been putting tripled quoted strings in our programs and functions?

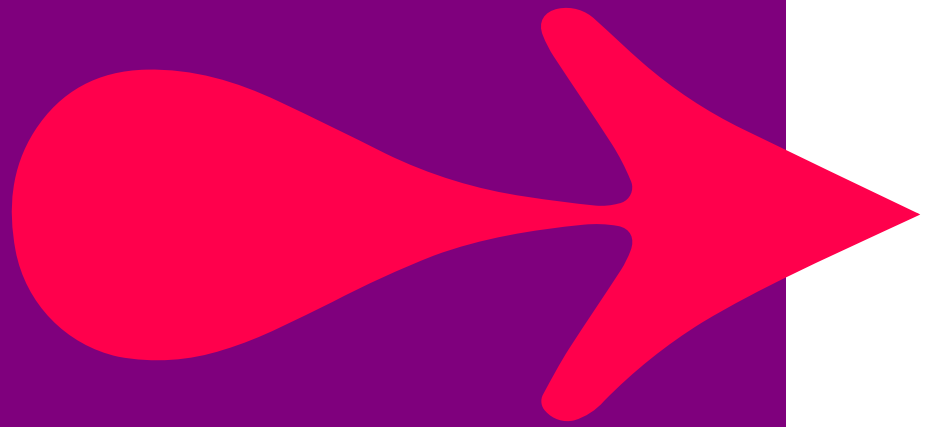
Well these are called DocStrings and usually appear immediately after the definition of a program/module, class, function or method.

Docstrings can be simple one-liners or multi-line and should not be descriptive (that is for # comments). Rather they must follow "Do this, return that" format. Multi-line dosctrings can be more elaborate, and are described in PEP 257, and can include details about input parameters and return values. Docstrings for classes (later in LIVE) should summarise its behaviour and list public methods. The __doc__ attribute can be used to access docstrings.

And they are read by the Python help() function - so now you know where we are getting help from for our modules and functions!

Docstrings

Trainer demonstration



demo_user_functions_docstrings.py

Learning check

5-10 mins

Quiz!

1. **What does (*,) at the beginning of the parameter list do?**
2. **What is the output of the following function call?**

```
>>> def my_func(num):
>>> return num + 10
>>>
>>> my_func(10)
>>> print(num)
```

3. **True of False. Do Python functions always return a value?**
4. **What is the output of the following function call?**

```
>> def display_info(**kwargs):
>>> for z in kwargs:
>>> print(z)
>>> display_info(movie="braveheart", year="1995")
```

Solutions

Functions quiz

1. What does (*,) at the beginning of the parameter list do?

   Answer: Enforce named parameter passing.

2. What is the output of the following function call?

```
def my_func(num):
    return num + 10

my_func(10)
print(num)
```

   Answer: NameError: name 'num' is not defined

3. True of False. Do Python functions always return a value?

   Answer: True

   (Python functions always return the value None unless explicit value given)

4. What is the output of the following function call?

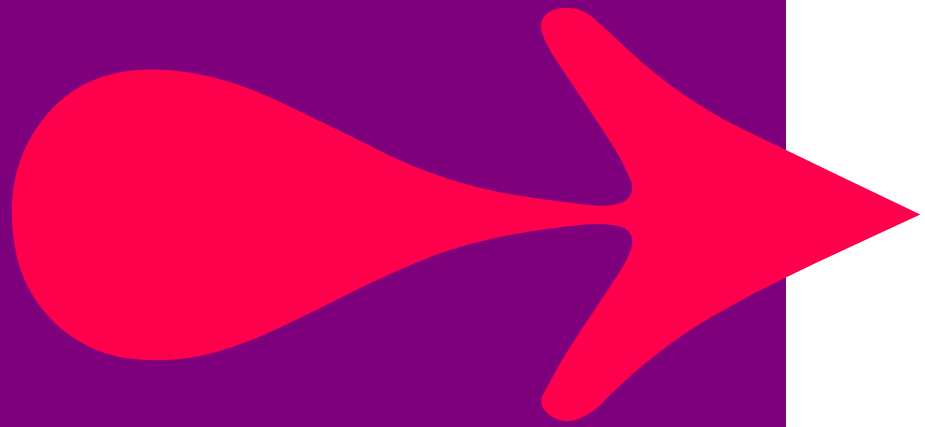
```
def display_info(**kwargs):
    for z in kwargs:
        print(z)
display_info(movie="braveheard", year="1995")
```

   Answer:

   movie

   year

Labs



1. Write your own calculator program called '**C:\lab\calc.py**' with add, divide, multiply, subtract and modulus functions.

Test your functions by passing in the following parameters.

add(11, 5)

divide(11,5)

multiply(11, 5)

subtract(11, 5)

modulus(11, 5)

2. You did add Docstrings to your functions? If not, add appropriate Docstrings to each function. To test, start IDLE and load your script (File/Open), then run (<F5>). Then in Python Shell type:

>>> help(add)

>>> help(multiply)

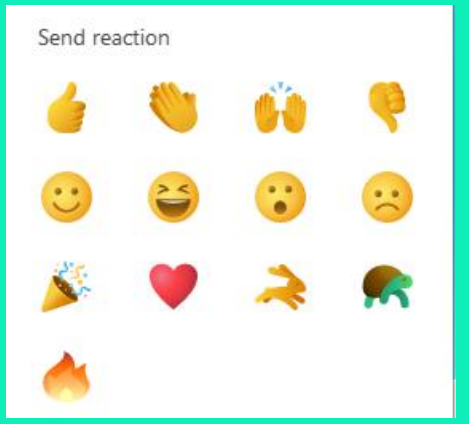
3. Modify the **calc.py** script to allow the add and multiply functions to accept a variable number of parameters. Test the modified functions by passing in the following parameters.

add(11, 5, 33, 15, 6.5)

multiply(11, 5, 33, 15, 6.5)

Stretch Exercises 4-5

END OF SECTION



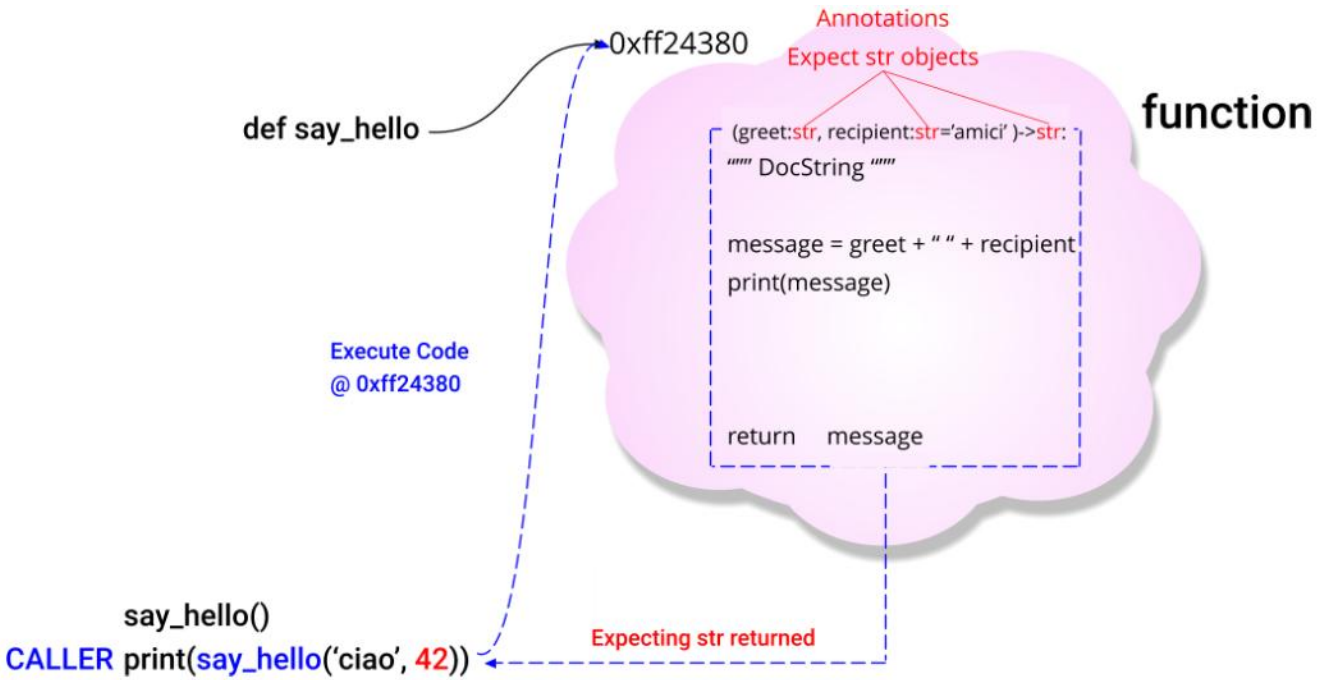
- An introduction to code reuse in Python
  - Functions
  - Parameters
  - Variadic Functions
  - Scope
  - Docstrings
- To be able to make code sharable and re-useable

Extension materials

Annotations

Annotations

- demo_user_functions_annotations.py



Function annotations were introduced in PEP 3107. They are used to provide additional information about the function, including what data types are expected for the input parameters and return value. They are like an embedded comment, but are more than a comment (which are ignored by the compiler). They are preserved in the function attribute __annotations__.

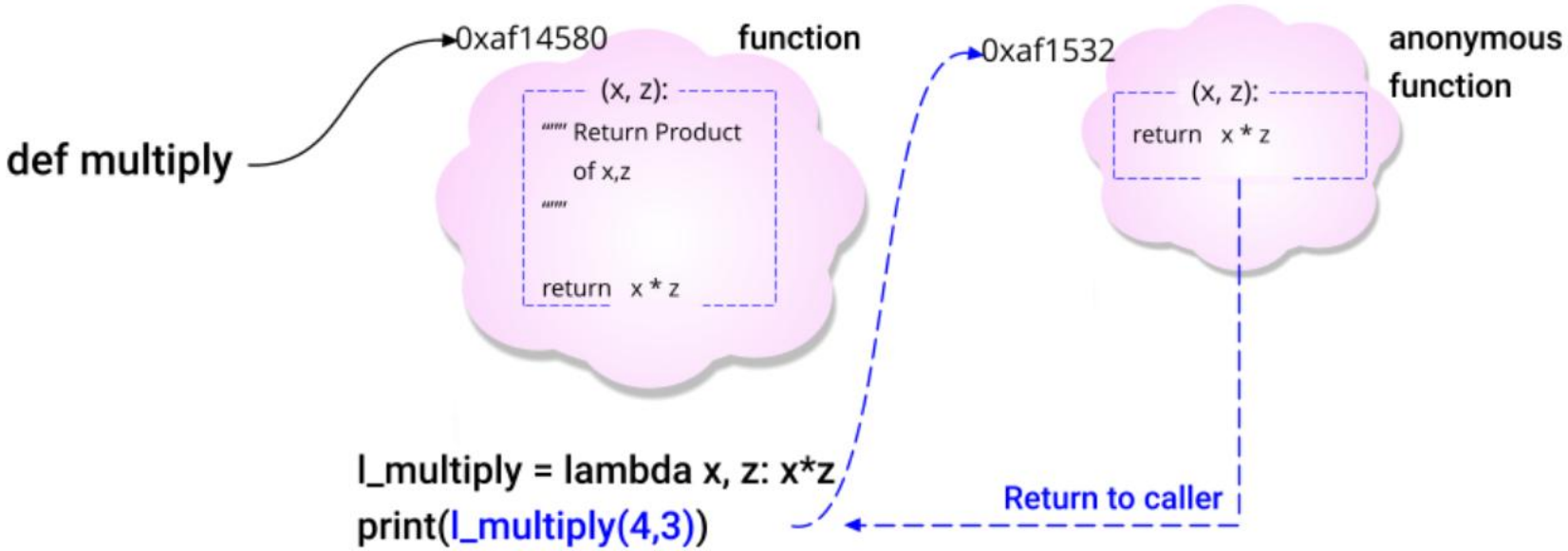
Annotations have no meaning except 'commenting in-code' of what types of data are hoped for. But 3rd party modules might use them for additional docstrings or for enforcing data type on parameters or return values. Python will remain a dynamically types language!

Extension materials

Lambda functions

Lambda Functions

- demo_user_functions_lambda.py

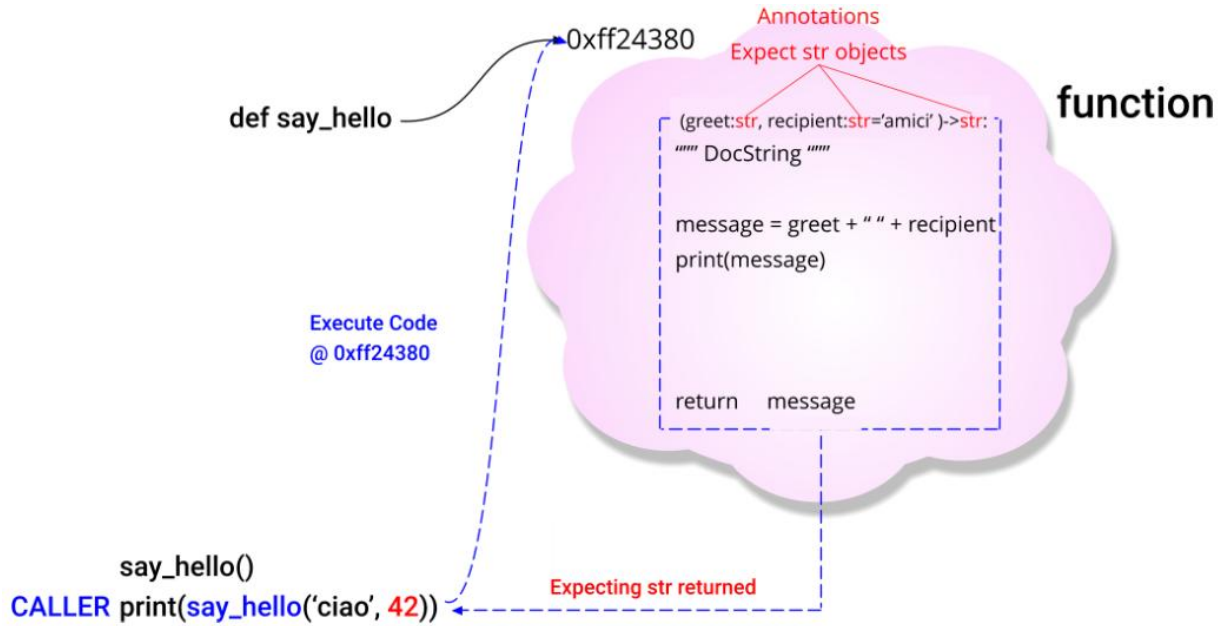


Extension materials

Nested functions

Nested Functions

- demo_user_functions_nonlocal.py



Python allows the nesting of functions inside each other, which not all languages support. It can be useful for simple reusable, recursive functions, function factories (a function can be returned from a function like any object) or for closures.

Like nested objects, functions follow the same scope rules. Nested functions can only be called within the function they are defined.

Python has a clean, safe, and dynamic way of managing variables/objects. As soon as you make an assignment, a new object is created - either a **GLOBAL** or a **LOCAL** (within a function). This mostly works well, but on occasion we might have nested functions, and we don't quite want to create a new variable and don't want to access the global variable either. This is where the **nonlocal** keyword proves useful to indicate the variable is referring to an enclosing (outer) scope variable.

REMINDER: TAKE A BREAK!

10.30 - 10.40
11.40 – 11.50
12.50 – 13.30
14.30 – 14.40
15.40 – 15.50