PYTHON FUNCTIONS

CSC 109 - Introduction to programming in Python - Summer 2023

Marcus Birkenkrahe

June 7, 2023

Contents

1 Python Functions: Simplifying Your Code

Why functions?

- Break down code into smaller, reusable pieces
- Make it easier to manage and maintain code
- Encapsulate tasks and variable within functions
- Call function from anywhere

What are functions?

- `def` keyword
- Name of function

Figure 1: Llyfrgell Genedlaethol Cymru / Claerwen Dam (1952)

- Pair of parentheses `()`
- Parameters of function
- Separator :
- Body of the function indented in the next line
- Return values anywhere in the function (`return`)

2 `def` Statements without Parameters

- Example 1: 'hello world' as a function without arguments:

```
# function definition
def helloWorld():
    print('Hello, world!')
    # function call
helloWorld()
```

- Example 2: What will the output of this script be if you call `howdy` three times in a row?

```
# function definition
def howdy():
    print('Howdy!')
    print('Howdy!!')
    print('Hello there.')
    # function calls
howdy()
howdy()
howdy()
```

```
Howdy!
Howdy!!
Hello there.
Howdy!
Howdy!!
Hello there.
Howdy!
```

```
Howdy!!
Hello there.
```

- You can view the execution of the program at autbor.com/hellofunc/

3 `def` Statement with Parameters

- You can define your own functions with *parameters*. When you pass *values* to the function, these are called *arguments*.
- Example 3: 'hello' as a greeting with `name` input:

```
def hello(name):
    print('Hello, ' + name)
```

- In this example, the function `hello` takes a parameter called `name`. When the function is called with a name, it prints a greeting message using that name.

```
hello('Marcus')
hello('Alice')
```

- When the function returns from being called, the value stored in a parameter is forgotten: what's the output of this script given the definition of `hello(name)` above?

```
hello('Bob')
hello(name)
```

4 Define, Call, Pass, Argument, Parameter, Recursion

- It may sound trivial, but it's not trivial to keep these concepts apart:

TERM/COMMAND	MEANING
Function definition	Create a function with `def` [name]([args]):
Function call	Executing function (with/out passing arguments)
Function argument	Value passed to a function in a function call
Function parameter	Variables that have arguments assigned to them

- Analyse this function and decide how to call it - what is 'result'?

```
def result(result):
    print(result)
```

- Example calls:

```
result(12)
result(12 + 500)
result('a')
result('hello world')
result('hello ' + 'world')
```

- What's what:
  1. `result` is a function name
  2. `result` is a parameter of the function `result`
  3. `result` is an argument of the function call `print`
- Can you call `result` inside `result`? (pythontutor.com)

```
def result(result):
    print(result)
    result(1)
    result(2)
```

  - The `TypeError: 'int' object is not callable` is because at that point, `result` has been redefined as a parameter of the function, not the function itself.
  - When you try to call `result(1)` inside the function, you treat `2` (the value passed as an argument to the function parameter) as a function - but `2` is an `int` and therefore not callable.

- A clearer version of this procedure:

```
def result(parameter):
    print(parameter)
    parameter(1) # Here parameter is not a function, it's the value
    # you passed (2)
result(2)
```

- How can you make a *recursive* function that calls itself? (PythonTutor)

```
def result(parameter):
    print(f'Parameter: {parameter}')
    if parameter > 0:  # a base case to stop recursion
        result(parameter - 1)  # call function itself, not the parameter
        result(2)
```

5 Practice defining functions with one parameter

1. In Colab, write a function `count` that takes a string `str` as an argument and prints the number of its characters.

   Tip: remember that there is a built-in function called `str.count` that can count the characters of a string `str` when given the right argument.

2. Call `count` on these arguments: `a`, `abcd`, `a b c d`. Output:

3. Is it Okay to call this function `count`?

   - It's OK to call your own function by a name used by Python: it will not affect the built-in function of the same name.
   - However, in your current scope (i.e. your Python session), it will overshadow the built-in function.
   - It is considered poor practice to re-use function names. In a modern editor, the syntax highlighting will tip you off.

4. Is it Okay to call the function parameter `str`?

   - It is OK to call a parameter inside your own function by a known name - it won't affect its use outside of the function.
   - However, inside the function, your name will overshadow the previous name used by Python.
   - It is considered poor practice to re-use keywords as names. In a modern editor, the syntax highlighting will tip you off.

5. Solution 1 (here in pythontutor with poor naming practice):

```
# function def
def cnt (string):
    print(string.count('')-1)

# function call
cnt('a')
cnt('abcd')
cnt('a b c d')
cnt(string='a b c d') # keyword parameter call
print('abcd'.count('')-1)  # standard Python `str.count` function
```

6. Solution 2:

```
def cnt1(string):
    return len(string)
print(cnt1('abcd'))
print(cnt1('a b c'))
```

```
4
5
```

6 Return Values and `return` Statements

- Functions can also provide an *output* or *return value* using the `return` statement. It consists of:
  1. the `return` keyword
  2. the value or expression that the function should return.
- The `return` statement causes the function to exit.
- Example 1 (can you identify the terms?):

```
def getAnswer(answerNumber):
    if answerNumber == 42:
        return 'The meaning of life, the universe, and everything.'
```

- Analysis of the function:
  1. `getAnswer` is a function

2. It takes a parameter `answerNumber`
3. The function checks if parameter is equal to `42`
4. If the parameter is equal to `42`, it returns a string.
5. If the parameter is not equal to `42`, it returns `None`.

- Let's check this out in pythontutor.

7 Extended example: 'Magic 8 Ball'

- Enter this code in Colab (without comments), then run it a few times (pythontutor):

```
import random                   #1

def getAnswer(answerNumber):    #2
    if answerNumber == 1:       #3
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidely so'
    elif answerNumber == 3:
        return 'It is Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
    elif answerNumber == 6:
        return 'Concentrate and ask again'
    elif answerNumber == 7:
        return 'My reply is no'
    elif answerNumber == 8:
        return 'Outlook not so good'
    elif answerNumber == 9:
        return 'Very doubtful'

r = random.randint(1,9)         #4
fortune = getAnswer(r)          #5
print(fortune)                  #6
```

- Analysis:

1. Import `random` module for random number functions.
2. Store a random integer from [1,9] in `r`.
3. Call function `getAnswer` with argument `r`.
4. Store `return` value from function in `fortune`.
5. `print` the `fortune`.
6. When calling a value outside of [1,9], `None` is `return` value.

- Introducing a `list` will allow us to shrink this script by a lot.

8 Practice using return values and statements

8.1 Calculate area of a rectangle

- Write a function called `calculate_area` that takes two parameters, `length` and `width`, and calculates the `area` of a rectangle. The formula to calculate the area of a rectangle is `area = length * width`. The function should return the calculated area via `print`.

- Test the function with the values (4,5) and (7,3) for (length,width), and the expected output 20 and 21, respectively.

- Sample solution:

```
# function definition
def calculate_area(length, width):
    area = length * width
    return print(area)
# function call
calculate_area(4,5)
calculate_area(7,3)
```

```
20
21
```

- What is the impact of `return`? Leave it out:

```
# function definition
def calculate_area_2(length, width):
    area = length * width
```

```
    print(area)
    # function call
calculate_area_2(4,5)
calculate_area_2(7,3)

20
21
```

- What is the impact of `print`? Return only the result:

```
# function definition
def calculate_area_3(length, width):
    area = length * width
    return area
# function call
print(calculate_area_3(4,5))
print(calculate_area_3(7,3))

20
21
```

8.2 Identify an even number

- Write a function called `is_even` that takes a single parameter, `number`, and checks if the number is even. If the number is even, the function should return `True`; otherwise, it should return `False`.
- Tip: to check if a number N is even, you can use the modulus operator `%` - the modulus of any even number with 2 is zero.
- Test the function with the values 4 and 7.

9 The `None` Value

- In Python, `None` represents the absence of a value.
- `None` is the only value of the `NoneType` data type (show this)[1].

[1] In R, missing values are indicated by the `NA` special value, which is of data type `logical` (aka Boolean). The `pandas` library in Python, missing values are represented as `NaN` (Not a Number). Both languages have many methods to deal with missing values, which are a frequent problem in real datasets.

```
print(type(None))
```

- `None` is used e.g. as the `return` value for `print()`:

```
spam = print('Hello') # prints 'Hello'
print(None == spam)   # spam now contains None
```

- Python adds `return None` to the end of any function definition with no `return` statement.
- This is similar to how a `while` or `for` loop implicitly ends with a `continue` statement (adding it makes the code more readable).
- Also, using `return` without a return value, returns `None`. Show this with a function that you write yourself!

```
def none():
    return
print(none())
```

10 Practice the `None` value

10.1 Maximum value in a list

- Write a function called `find_max` that takes a list of numbers as a parameter and returns the maximum value in the list. If the list is empty, the function should return `None`.
- Tip: you can use the built-in function `max` to identify the maximum number in a Python list, or you can devise your own algorithm (home bonus assignment).
- Test the function by calling it with these sample arguments:

```
print(find_max([2, 4, 6, 8, 10]))  # Output: 10
print(find_max([]))  # Output: None
```

10.2 Check divisibility of two integers (do it at home)

- Write a function called `check_divisibility` that takes two integers, `num` and `divisor`, as parameters. The function should check if `num` is divisible by `divisor` without a remainder. If it is divisible, the function should return `True`; otherwise, it should return `None`.

- Tip: to check if a number N is divisible by a number M, you can use the modulus operator `%` - the modulus of N and M is zero if they are divisible.
- Test the function with different arguments:

```
print(check_divisibility(10, 5))  # Output: True
print(check_divisibility(10, 7))  # Output: None
```

11 Keyword Arguments and `print()`

- Arguments are either positional arguments or keyword arguments
- Positional arguments are identified by their position only
- Keyword arguments can be assigned default values
- The `print` function is an example:

```
print('Hello', end='')
print('World')
```

- Which other keyword parameters does `print` have?

```
print(help(print))   # on IPython, use `print?`

Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.

None
```

- Print `Hello, World Hello, World` using only 'Hello' and 'World':

```
print('Hello', 'World', end=' ', sep=', ')
print('Hello', 'World', end='', sep=', ')

Hello, World Hello, World
```

- Print 'Hello, World!' to a file named `helloworld.txt`, then check if the file was created with 'magic' IPython commands `%ls` and `%cat`:

```
# tell computer to write stdout to a file f
with open('helloworld.txt', 'w') as f:
    print('Hello, World!', file=f)

# in IPython, list file and view content
%ls -l helloworld.txt
%cat helloworld.txt
```

- The `flush` keyword parameter default is `False`, which means that the output is buffered (held) before being written to stdout.
- When you want logging or status messages during run-time to be directly visible, `flush=True` might be useful:

```
import time

for i in range(5):
    print(i, end=' ', flush=True)  # write i immediately
    time.sleep(1) # pause for 1 second
```

- It makes sense to spend some time experimenting with the keyword parameters of important built-in functions that you use a lot.
- You can add your own keyword arguments to the functions as well (after learning more about lists and dictionaries).

12 The Call Stack

- A conversation could be called 'stack-like' if the current topic is always at the top of the stack structure:

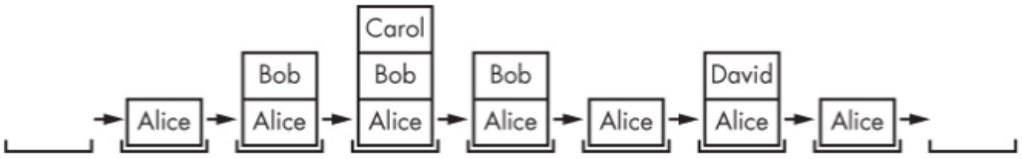

Figure 2: Conversation as call stack

- Similarly, Python remembers which line of your script called the function and will return there when it hits a `return` statement.
- If that function called other functions, it would return to those functions first before returning to the original function call.
- Check out this program (autbor.com/abcdcallstack/):

```
def a():
    print('a starts')
    b()
    d()
    print('a returns')

def b():
    print('b starts')
    c()
    print('b returns')

def c():
    print('c starts')
    print('c returns')

def d():
    print('d starts')
    print('d returns')

# function call
a()

a starts
b starts
```

```
c starts
c returns
b returns
d starts
d returns
a returns
```

- The stack picture looks like this:

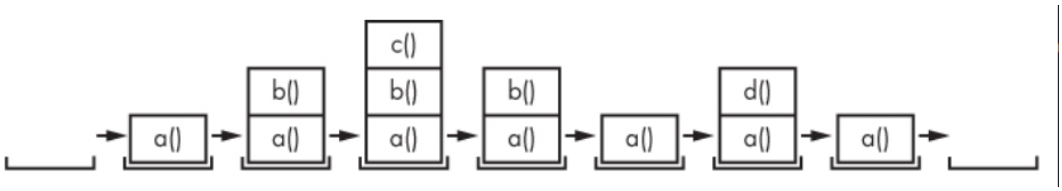

Figure 3: abcd call stack

- Frame objects are only added and removed from the top of the stack.
- The top of the stack is which function is currently being executed.

13 Local and Global Scope

- Variables that are assigned in functions are in *local scope* - they are only known (and can be used only) inside the function.
- Variables that are assigned outside of any function are in *global scope* - they are known (and can be used) anywhere in the script.
- A variable must be either local or global in scope.
- Scopes are like containers: When a scope is destroyed, all the values stored in the scope's variables are forgotten:
  1. When a function call is ended, local scope is destroyed.
  2. When a program is finished, global scope is destroyed.
- Why is this important?
  1. Code in global scope cannot use local variables
  2. Code in local scope can use global variables

3. You can use the same name for variables in different scopes
4. This narrows the number of lines that could cause a bug.

14 NEXT Local and global variables with the same name

- Check out this example (in the pythontutor you can see the frames): how many variables called `eggs` are there?

```
def spam():
    eggs = 'spam local'
    print(eggs) # prints 'spam local'

def bacon():
    eggs = 'bacon local'
    spam()   # call spam
    print(eggs)  # prints 'bacon local'

eggs = 'global'  # global `eggs`
bacon()          # local `eggs` in bacon() and spam()
print(eggs)      # global `eggs`
```

- **Analyze this:** which printout do you expect - will this work?

```
# function definition
def hello1():
    print('Hello from hello1')
    def hello2():
        return print('Hello from hello2')
    hello2()
    # function calls
hello1()
hello2()
```

- **Analysis:**

1. All functions are void (take no arguments).
2. `hello1` prints message and calls `hello2`
3. `hello2` prints message and returns `print` message

4. `hello1` returns `None` and `hello2` is destroyed
5. `hello2` is called outside of scope - `NameError`

15 Practice local scope

- Write a function called `add` that takes two parameters, `x` and `y`. Inside the function, declare a local variable called `z` and assign it the sum of `x` and `y`. Print the value of `z` inside the function. Then, outside the function, print the value of `z`.

16 The `global` statement

- Use `global` to modify a global variable from within a function: the line `global eggs` at the top of a function says to Python "don't create a local variable with this name!"
- View the program execution at autbor.com/globalstatement/

```
def spam():
    global eggs
    eggs = 'spam' # this is now the global value of `eggs`

eggs = 'global'
spam()  # returns the global value of `eggs`
print(eggs)
```

- There are four rules to tell which scope a variable is in:
  1. If is used in the global scope (outside of all functions), then it is always a global variable.
  2. If there is a `global` statement in a function, it is a global variable.
  3. If there is no `global` statement and the variable is used in an assignment in the function, it is a local variable.
  4. But if the variable is not used in an assignment statement, it is a global variable.
- Identify output and local or global variables (pythontutor):

```
def spam():
    global eggs
    eggs = 'spam'

def bacon():
    eggs = 'bacon'

def ham():
    print(eggs)

eggs = 42
spam()
print(eggs)
```

- Identify output and local or global variables (pythontutor):

```
count = 0

def cnt():
    count = 0
    return count

def increment():
    global count
    count += 1

cnt()
increment()
print(count)
```

17 Referencing local variables before assignment

- If you try to use a local variable in a function before you assign a value to it, you get an `UnboundLocalError` (pythontutor):

```
def spam():
    print(eggs) # ERROR
    eggs = 'spam local'

eggs = 'global'
```

```
spam()

print(eggs) # ERROR
    ^^^^
UnboundLocalError: cannot access local variable 'eggs'
where it is not associated with a value
```

- Python sees the assignment for `eggs` in the function and therefore considers it local.
- But when trying to execute `print(eggs)`, `eggs` does not exist, and Python will not fall back to using the global `eggs` variable.

18 Practice the `global` statement

- Write a *void* function called `modify_global_variable` that takes no parameters. Inside the function, declare a `global` variable called `count` and assign it an initial value of `0`. Increment the value of `count` by `1` using an *augmented assignment* operator `+=`. Print the value of `count` inside the function. Then, outside the function, increment the value of `count` by `1` and print it. **Print all statements with f-strings**.
- Sample output:

```
Inside the function: 1
Outside the function: 2
```

- Copy your solution to pythontutor.com to visualize the execution.

19 Exception handling with `try` and `except`

- In real-world programs, you want Python to detect errors, handle them, and continue to run.
- Example: this program has a fatal divide-by-zero error.

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
```

```
print(spam(12))
print(spam(0))
print(spam(1))
```

- The error name is `ZeroDivisionError`. From the traceback, you know that the `return` statement is causing the error.
- To handle this exception, put the divide-by-zero code in a `try` clause and add an `except` clause to handle the error scenario:

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

```
21.0
3.5
Error: Invalid argument
None
42.0
```

- Why is `None` printed out?

  Answer: because the `except` clause does not end with a `return` statement.

- Any errors that occur in function calls in a `try` block will be caught (see pythontutor):

```
def spam(divideBy):
    return 42 / divideBy

try:
    print(spam(2))
    print(spam(12))
```

```
    print(spam(0))
    print(spam(1))

except ZeroDivisionError:
    print('Error: Invalid argument')
```

20 `try` harder `except` when you're `finally` done

- You can add as many `except` statements as you like, for more than one error. Example: a version of `float` that fails gracefully.
- Remember: executing the `except` clause means that a legitimate error is suppressed and the program fails in a controlled manner only.
- The `except` clause will only be executed if `float(x)` raises an exception (in this case a `ValueError`):

```
def attempt_float(x):
    try:
       return print(float(x))
    except:
       return print(f"Cannot convert '{x}' to float.")

attempt_float('1.2345')
attempt_float('something')   # ValueError
```

- A `float` conversion can also raise a `TypeError`:

```
float((1,2))  # TypeError: cannot convert a tuple
```

- Here, the `except` clause will only be executed if a `ValueError` is raised (the `TypeError` might indicate a legitimate bug):

```
def attempt_float(x):
    try:
       return print(float(x))
    except ValueError:
       return print(f"ValueError: cannot convert '{x}' to float.")

attempt_float((1,2))  # TypeError not excepted - program terminates
attempt_float('1.2345')
attempt_float('something')   # ValueError is excepted
```

- The `except` clause will only be executed if a `ValueError` is raised:

```
def attempt_float(x):
    try:
       return print(float(x))
    except ValueError:
       return print(f"V: cannot convert '{x}' to float.")

attempt_float('1.2345')
attempt_float('something')   # ValueError
```

- You can catch multiple exception types simultaneously, too:

```
def attempt_float(x):
    try:
        return print(float(x))
    except (TypeError, ValueError):
        return print(f'TypeError or ValueError: {x}')
attempt_float('a')
attempt_float((1,2,3))
attempt_float(1)

TypeError or ValueError: a
TypeError or ValueError: (1, 2, 3)
1.0
```

- Or you can catch them serially, like this:

```
def attempt_float(x):
    try:
        return print(float(x))
    except TypeError:
        return print(f'TypeError: {x}')
    except ValueError:
        return print(f'ValueError: {x}')

attempt_float('a')
attempt_float((1,2,3))
attempt_float(1)
```

```
ValueError: a
TypeError: (1, 2, 3)
1.0
```

- You may not want to suppress an exception but code to be executed regardless - use `finally` for that:

```
f = open(path, mode='w') # open path to file f
try:
    write_to_file(f)  # write to file f
finally:
    f.close()          # file f will ALWAYS get closed
```

- You can have code that executes only if the `try` clause succeeds using `else` and `finally`:

```
f = open(path, mode='w') # open path to file f
try:
    write_to_file(f)  # write to file f
except:
    print('Failed')   # exit if writing didn't work
else:
    print('Succeeded')  # exit if writing did work
finally:
    f.close()   # close f no matter what
```

21 Practice Exception Handling

- Write a function that takes two arguments `a` and `b` and returns their sum.
- Handle the potential error when trying to add a string or a number.
- To test the function, use the following testdata:

```
prt(1,2)
prt('hello','world')
prt('hello',1)
```

22 Practice Exception Handling (home/bonus)

- Write a function `div` that asks the user to enter two numbers `num1` and `num2` (with `input`) and divides the first number by the second number.
- Inside the function, handle two potential exceptions: 1) division by zero, and 2) invalid input (
- Tip: do this with `try` and `except` and check out which errors are raised when dividing by zero or providing invalid input.
- Tip: You can stack `except` clauses like shown here:

```
try:
   [do something]
except ErrorType1:
   [print something]
except ErrorType2:
   [print something]
```

- Sample input/output session:

```
>>>
Enter first number: 2
Enter second number: 3
The result is 0.6666666666666666
>>> div()
Enter first number: 10
Enter second number: 5
The result is 2.0
>>> div()
Enter first number: 1
Enter second number: 10
The result is 0.1
>>> div()
Enter first number: 1
Enter second number: 0
Division by zero is not allowed
>>> div()
Enter first number: 10
Enter second number: a
Invalid input. Please enter a number.
>>>
```

23 Short program: Zig-zag

- This program will create a back-and-forth, zig-zag pattern until the user stops it by pressing `CTRL-c`. See here for a notebook in GitHub.
- Sample output:
- Type this code into Colab:

```
import time, sys
indent = 0
```

Figure 4: Sample output from zig-zag program

```
indentIncreasing = True

try:
  while True:
      print(' ' * indent, end='')
      print('********')
      time.sleep(0.1)

      if indentIncreasing:
          indent += 1
          if indent == 20:
              indentIncreasing = False
      else:
          indent -= 1
          indentIncreasing = True

except KeyboardInterrupt:
    sys.exit()
```

- Analysis:

  1. The program begins with importing two modules: `time` for time-keeping, and `sys` for the `exit` function, which triggers a `KeyboardInterrupt` 'error':

     ```
     import time, sys
     ```

  2. Two loop variables are `indent` for the number of spaces to indent per line, and `indentIncreasing`, a Boolean variable that indicates direction: `True` for moving to the right, `False` for the left.

     ```
     indent = 0
     indentIncreasing = True
     ```

  3. The rest of the program is placed in a `try` statement: do whatever follows unless the `except` condition is triggered. The script enters an *infinite loop* to print `indent` number of spaces next to one another followed by eight asterisks. The script halts for 1/10 secs. after the print.

     ```
     try:
       while True:
     ```

```
        print(' ' * indent, end='')
        print('********')
        time.sleep(0.1)
```

4. To adjust the indentation until the asterisks are printed, we check if `indentIncreasing` is `True`: if it is, we indent until the indentation hits the value `20`, and switch `indentIncreasing` to `False`:

```
if indentIncreasing:
    indent += 1
    if indent == 20:
        indentIncreasing = False
```

5. If `indentIncreasing` is `False`, the `else` condition is true and we reduce the indentation in `indent` by one until we hit `0`. Then we switch direction by setting `indentIncreasing` to `True`.

```
else:
  indent -= 1
    indentIncreasing = True
```

6. After checking the conditions, the program goes back to the start of the infinite loop and executes again. If the user triggers a keyboard interrupt with CTRL-c (or CTRL-m + i in Colab, or by pressing the STOP button next to the code cell), `sys.exit` is executed, the loop is left and the program is finished.

```
except KeyboardInterrupt:
    sys.exit()
```

24 Summary

- Functions provide a way to encapsulate reusable code blocks, accept inputs through parameters, and return outputs using return statements.
- Understanding how to define and use functions effectively will enhance your code organization, reusability, and overall readability.
- Local and global scope helps you encapsulate and isolate values for program writing, testing and debugging.
- Exception handling statements run code when a specific error has been detected to make your programs more resilient to common errors.

25 Glossary

TERM/COMMAND	MEANING
Function definition	Create a function
Function call	Executing function (with/out passing arguments)
Function argument	Value passed to a function in a function call
Function parameter	Variables that have arguments assigned to them
Keyword parameter	Paramater optionally called with a name
Positional parameter	Parameter called with position only
Recursive function	Function that calls itself
`None`	Value that indicates a missing value
Return value	Value that is returned by a function
Positional argument	Value in function call for positional parameter
Keyword argument	Value in function call for keyword parameter
Local scope	Variables known only in functions
Global scope	Variables known everywhere in the script
Void function	Function without parameters like `print()`
`try:...except:`	Exception handling

26 References

- Sweigart, A. (2019). Automate the Boring Stuff with Python. NoStarch. URL: automatetheboringstuff.com