OReilly - 2024-02February-29

March 7, 2024

1 Agenda: Day 4 – functions

1. Questions
2. What are functions?
3. How do we define functions?
4. Arguments and parameters
5. Return values (return values vs. printing)
6. Default argument values
7. Complex return values (and some unpacking)
8. Local vs. global variables

2 What are functions?

We've been using functions (and methods) throughout this course:

- `len`
- `print`
- `input`
- `str.strip`
- `str.split`

Functions are the verbs of a programming language; they get things done.

You might remember the DRY ("don't repeat yourself") rule, which says:

- If we have the same code several times in a row, we can "DRY it up" into a loop
- If we have the same code in several different places in our program, we can "DRY it up" into a function. That is: We define the function in one place, and then use it in many other places.

Defining functions gives us many practical and semantic advantages: - We only have to write the code a single time. - When it comes time to debug/improve/change/optimize our code, having things in a function makes that so much easier - We gain semantic power by using a function, creating a higher level of abstraction.

Abstraction is the idea that we can take some functionality, wrap it up under a name, and not think about the underlying implementation. We can treat it as a black box that accomplishes a goal. Once we've created that abstraction, we can use it in higher-level things.

By writing functions, we create higher-level abstractions. We can think at a higher level, and solve bigger problems by ignoring the nitty-gritty that's going on inside of the function. A great deal of programming involves writing functions so that we can think at that higher level.

3 How do I define a function?

- We write `def`, a reserved word
- We give the function a name – it's a variable name, so it has to follow variable-name rules
- After the name, we put `()`, currently empty, but we will fill them with parameters in the coming hour
- The line ends with a `:`
- Following a `:`, we always have an indented block in Python. This is known as the "function body," and it is what executes every time we invoke the function. You can put whatever code you want inside of the function body – `if`, `input`, `print`, `for`, `while`…
- When the block ends, the function definition ends, too.

[1]:
```
def hello():
    print('Hello!')
```

[2]:
```
# every time I want to print hello on the screen, I just run the hello()␣
 ↪function

hello()
```

```
Hello!
```

[3]:
```
# once I have the function defined, I can use it inside of other constructs

for i in range(3):
    hello()
```

```
Hello!
Hello!
Hello!
```

4 Exercise: Simple greeting

Define a function, `greet`, that when run asks the user to enter their first and last names (separately, assigned to two variables), and then prints a nice greeting that uses both of their names.

Example:

```
greet()
Enter your first name: Reuven
Enter your last name: Lerner
Hello, Reuven Lerner!
```

[1]:
```
def greet():
    first_name = input('Enter first name: ').strip()
    last_name = input('Enter last name: ').strip()
    print(f'Hello, {first_name} {last_name}!')
```

[2]:
```
greet()
```

```
Enter first name:  Reuven
Enter last name:  Lerner

Hello, Reuven Lerner!
```

5 Arguments and parameters

When we call a function, we can pass values to it. These values are known as *arguments*. We've done this many times already:

```
len('abcd')   # here, 'abcd' is the argument
len([10, 20, 30])  # here, [10, 20, 30] is the argument
print('hello')  # here, 'hello' is the argument
```

In order for our function to accept arguments, we will need to redefine it such it has one or more *parameters*. Meaning: It'll need to have variables into which the arguments will be assigned.

You can think of parameters as variables that get their values from whoever calls the function; you never need to worry about defining their values yourself.

Almost every programmer I know confuses the terms "arguments" and "parameters." If you use one for the other, that's mostly OK. But you should try to keep them separate:

- Arguments are values
- Parameters are variables

Arguments are assigned to parameters when we call a function.

[3]:
```
# let's rewrite our "hello" function

def hello(name):    # name is a parameter, which will get the value of whatever␣
 ↪argument we pass
    print(f'Hello, {name}!')
```

[5]:
```
# parameters:   name
# arguments:   'world'

hello('world')    # positional argument -- it is assigned to a parameter based␣
 ↪on their positions
```

```
Hello, world!
```

[6]:
```
# what if I now try to call hello without any argument?

hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 3
      1 # what if I now try to call hello without any argument?
----> 3 hello()
```

```
TypeError: hello() missing 1 required positional argument: 'name'
```

In Python, defining a function includes assigning the new function to a variable.

Just as a variable cannot refer to both 5 and 7 at the same time, a variable cannot refer to two different versions of the same function at the same time.

The most recent assignment to a variable (with `=` or with `def`) is the one that currently stands.

[7]:
```
# let's rewrite our "greet" function to take arguments

def greet(first_name, last_name):   # notice -- two parameters, separated by␣
 ↪commas
    print(f'Hello, {first_name} {last_name}!')
```

[8]:
```
# parameters: first_name   last_name
# arguments:   'Reuven'      'Lerner'

greet('Reuven', 'Lerner')
```

```
Hello, Reuven Lerner!
```

6 Exercise: Calculator

1. Define a function, `calc`, that takes three arguments:
   - an integer
   - a string, either `+` or `-`
   - another integer
2. If the user passed `+` or `-`, then show the full math expression, including a result
3. If not, then indicate that the operator wasn't known

Example:

```
calc(2, '+', 3)   # prints '2 + 3 = 5'
calc(20, '-', 5)  # prints '20 - 5 = 15'
calc(3, '*', 4)   # prints '3 * 4 = (unknown operator *)'
```

[9]:
```
def calc(first, op, second):
    if op == '+':
        result = first + second
    elif op == '-':
        result = first - second
    else:
        result = f'(unknown operator {op})'

    print(f'{first} {op} {second} = {result}')
```

[10]:
```
# parameters:   first   op   second
# arguments:     10     '+'    3

calc(10, '+', 3)
```

```
10 + 3 = 13
```

[11]:
```
calc(200, '-', 150)
```

```
200 - 150 = 50
```

[12]:
```
calc(3, '*', 5)
```

```
3 * 5 = (unknown operator *)
```

[13]:
```
# remember hello?

hello('world')
```

```
Hello, world!
```

[14]:
```
# what happens if I call hello with an integer?
hello(5)
```

```
Hello, 5!
```

[15]:
```
hello([10, 20, 30])
```

```
Hello, [10, 20, 30]!
```

[16]:
```
# can I go completly bananas and call hello with a function argument?!?
hello(hello)
```

```
Hello, <function hello at 0x10e9ca200>!
```

7 Arguments and type checking

In Python, and other dynamic languages, there is no way to tell a variable that it can only contain a certain type of value. If you define `x` to be `'abcd'`, and then you set `x` to be `123`, that's totally fine.

Statically typed languages don't let you do this: You have to tell the language what kind(s) of values will be assigned to a variable. If it sees you assigning the wrong type, it gives you an error message – before you run the program.

There isn't any way to prevent someone from calling your function with the wrong kind of value. What can you do?

1. Document your functions (with docstrings)
2. Put in some checks in your functions (yuck)
3. Use a type checker in Python (e.g., Mypy), which we won't be discussing in this course
4. Have errors

5. Trap exceptions (which we won't be discussing in this course) – this is the most Pythonic way

[17]:
```
# SC tried to use the "isnumeric" or "isdigit" methods, but they didn't work on integers
# that's right -- they are string methods!

x = '123'
x.isdigit()
```

[17]:
```
True
```

[18]:
```
x = 123  # integer
x.isdigit()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[18], line 2
      1 x = 123  # integer
----> 2 x.isdigit()

AttributeError: 'int' object has no attribute 'isdigit'
```

[19]:
```
# if you really want, you can use isinstance

isinstance(x, int)
```

[19]:
```
True
```

In dynamic languages, we often talk about "duck typing" – meaning that we don't care what kind of value we get, but we do care what it does.

8 Next up

1. Docstrings (documenting our functions)
2. Keyword arguments
3. Return values

[23]:
```
# SC
# when you define a function inside of a function, it's known as an "inner function"

def foo():
    def fun():
        return 1
    return fun()
```

```
def bar():
    def fun():
        return 2
    return fun()
```

[21]:
```
foo()
```

[21]:
```
1
```

[22]:
```
bar()
```

[22]:
```
2
```

9 Documenting your function

So far, we've hoped/assumed that someone calling our function would know what the function does, and what kinds of arguments it takes. That's not a good strategy. We should document that somewhere.

The way that we document functions in Python is with "docstrings" – if the first line of a function is a string, then it is taken as the documentation.

Don't confuse this documentation with comments in the code!

- comments are for the people who will be maintaining the software
- docstrings are for people who will be using the software

[26]:
```
def hello(name):
    'This is the best function ever written!'
    print(f'Hello, {name}!')
```

[27]:
```
hello('world')
```

```
Hello, world!
```

[28]:
```
# in Jupyter, we can use the "help" function
# in most editors/IDEs, you can hover over a function name and get the docs.

help(hello)
```

```
Help on function hello in module __main__:

hello(name)
    This is the best function ever written!
```

[29]:
```
# what if we want a longer docstring?
# we can use triple-quoted strings, with """ and """ at the start and end
# this way, you can have more than one line

```

```
def hello(name):
    """Print a friendly greeting to the user.

    Expects: One argument, a string, which will be assigned to name
    Modifies: Nothing
    Returns: Nothing
    """
    print(f'Hello, {name}!')
```

[30]:
```
help(hello)
```

```
Help on function hello in module __main__:

hello(name)
    Print a friendly greeting to the user.

    Expects: One argument, a string, which will be assigned to name
    Modifies: Nothing
    Returns: Nothing
```

10 Return values

When we call a function like `len`, we expect to get a value back:

```
x = len('abcd')      # x will be 4
```

So far, none of our functions have returned values. They have displayed values on the screen, but they have not returned values that we could store in variables.

The difference between printing and returning is very big, but it's less obvious when you're using Jupyter, because returned values are automatically displayed.

- If a function returns a value, then that value can be assigned to a variable, passed to another function, or printed
- If a function prints a value, then that value is printed, and we can't do anything more with it.

Also: A function can print as much (or as little) as it wants. But it can only return one value per invocation of the function.

As a general rule, I encourage you to return values from your functions (not print them), but to print any debugging info that you might find useful.

How do we return a value? We use the `return` statement in Python. We can return any value we want, without any exceptions.

- If you invoke `return` without giving it a value, it returns the special value `None`
- If your function never calls `return`, then it also returns `None` when it's done

You can use parentheses with `return`, but we normally don't.

[31]:
```
def hello(name):
    return f'Hello, {name}!'
```

[32]:
```
hello('world')   # I'm running it in Jupyter, so I'll get a value back, and␣
↪it'll be displayed
```

[32]:
```
'Hello, world!'
```

[33]:
```
x = hello('world')
print(x)
```

```
Hello, world!
```

11 Exercise: Calculator

1. Write a new `calc` function that takes one string argument in the format of `'X op Y'`, where `X` and `Y` are digits and `op` is either `+` or `-`.
2. Break that string apart into pieces, and convert `X` and `Y` into integers.
3. Return the same kind of string as before, with `'2 + 3 = 5'` and if the operator isn't valid, indicate that in the result.

[40]:
```
def calc(s):
    fields = s.split()
    first = fields[0]
    op = fields[1]
    second = fields[2]

    first = int(first)
    second = int(second)

    if op == '+':
        result = first + second
    elif op == '-':
        result = first - second
    else:
        result = f'(Unrecognized operator {op})'

    return f'{first} {op} {second} = {result}'
```

[41]:
```
calc('2 + 3')
```

[41]:
```
'2 + 3 = 5'
```

[42]:
```
calc('20 - 3')
```

[42]:
```
'20 - 3 = 17'
```

[43]:
```
calc('20 * 3')
```

[43]:
```
'20 * 3 = (Unrecognized operator *)'
```

[46]:
```
def calc(s):
    # use unpacking
    first, op, second = s.split()

    if first.isdigit() and second.isdigit():

        first = int(first)
        second = int(second)

        if op == '+':
            result = first + second
        elif op == '-':
            result = first - second
        else:
            result = f'(Unrecognized operator {op})'

        return f'{first} {op} {second} = {result}'

    else:
        return f'Bad values -- not numeric!'
```

[47]:
```
calc('20 + 4')
```

[47]:
```
'20 + 4 = 24'
```

[48]:
```
calc('cat + dog')
```

[48]:
```
'Bad values -- not numeric!'
```

12 Keyword arguments

So far, all of the arguments we have passed to our functions have been *positional* arguments. However, there is another kind of argument, namely *keyword* arguments.

Positional arguments are assigned to their parameters based on their positions. Keyword arguments, by contrast, are assigned to parameters based on the names we pass. That is, we explicitly tell Python which parameter should be assigned which value, regardless of position.

All keyword arguments have the form of `NAME=VALUE`, with the `=` in the middle.

You can pass any number of positional arguments, and any number of keyword arguments – but all positional arguments must come before all keyword arguments.

[49]:
```
def hello(name):
    return f'Hello, {name}!'
```

[51]:
```
hello('world')  # positional argument
```

[51]:
```
'Hello, world!'
```

[52]:
```
hello(name='world')   #keyword argument
```

[52]:
```
'Hello, world!'
```

[53]:
```
def add(first, second):
    return first + second
```

[56]:
```
# parameters: first second
# arguments:   10     3
add(10, 3)
```

[56]:
```
13
```

[57]:
```
# parameters: first second
# arguments:    10     3
add(first=10, second=3)
```

[57]:
```
13
```

[58]:
```
# parameters: first second
# arguments:    3    10
add(second=10, first=3)
```

[58]:
```
13
```

[63]:
```
# parameters: first  second
# arguments    5       8

add(5, second=8)
```

[63]:
```
13
```

[60]:
```
add(first=5, 8)
```

```
  Cell In[60], line 1
    add(first=5, 8)
                 ^
SyntaxError: positional argument follows keyword argument
```

[61]:
```
add(second=5, second=8)
```

```
  Cell In[61], line 1
    add(second=5, second=8)
                  ^
```

```
SyntaxError: keyword argument repeated: second
```

[62]:
```
add(5, first=8)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[62], line 1
----> 1 add(5, first=8)

TypeError: add() got multiple values for argument 'first'
```

[64]:
```
add(x=100, y=200)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[64], line 1
----> 1 add(x=100, y=200)

TypeError: add() got an unexpected keyword argument 'x'
```

13 Exercise: `mysum`

Python comes with a function, `sum`, that takes a list of integers and returns their sum.

Write your own function, `mysum`, that takes a list of integers and returns their sum. Do *NOT* use the builtin `sum` function in your own function. Rather, think about how you can implement it, and do so.

Example:

```
mysum([10, 20, 30])           # should return 60
mysum(numbers=[10, 20, 30])   # should return 60
```

[65]:
```
def mysum(numbers):
    total = 0

    for one_number in numbers:
        total += one_number

    return total
```

[66]:
```
mysum([10, 20, 30])
```

[66]: 60

[67]:
```
mysum(numbers=[10, 20, 30])
```

[67]:
```
60
```

[68]:
```
# AB

# numbers = []
# while True:
#     numer = input('Type a number or exit if you want to stop: ')
#     if numer.isdigit():
#         numbers.append(int(numer))
#     else:
#         break
# print(numbers)

def mysum(num):
    total = 0
    for i in num:
        total += i
    return total

print(mysum([10, 20, 30]))
```

```
60
```

14 Next up

1. Complex return values
2. Local vs. global variables
3. Special parameters

[69]:
```
def get_numbers():
    return [10, 20, 30]

get_numbers()
```

[69]:
```
[10, 20, 30]
```

[70]:
```
x = get_numbers()   # this is fine
x
```

[70]:
```
[10, 20, 30]
```

[71]:
```
x,y,z = get_numbers()   # how is this fine? Unpacking!
```

[72]:
```
x
```

[72]:
```
10
```

[73]:
```
y
```

[73]:
```
20
```

[74]:
```
z
```

[74]:
```
30
```

[75]:
```
# if a function returns a complex data structure,
# we can use unpacking and similar techniques to grab it (or its parts)
```

[76]:
```
def get_status():
    return 200, {'text':'ok', 'url':'python.org'}
```

[78]:
```
get_status()
```

[78]:
```
(200, {'text': 'ok', 'url': 'python.org'})
```

[79]:
```
status_code, status_dict = get_status()  # the 2-element tuple we got back is␣
↪unpacked to two variables
```

[80]:
```
status_code
```

[80]:
```
200
```

[81]:
```
status_dict
```

[81]:
```
{'text': 'ok', 'url': 'python.org'}
```

[82]:
```
status_dict['text']
```

[82]:
```
'ok'
```

[83]:
```
status_dict['url']
```

[83]:
```
'python.org'
```

15 Exercise: Smallest and biggest

1. Write a function, `smallest_and_biggest`, that takes a single argument, a list of numbers.
2. The function will return a 2-element list, containing the smallest and largest number in the input list

Example:

```
smallest_and_biggest([10, 5, 3, 17, 12])   # returns [3, 17]
```

- Hint: Assume that the first number in the list is both the biggest and the smallest
- Then iterate over each element, comparing it with the current biggest and smallest values

[84]:
```
def smallest_and_biggest(numbers):
    # setup
    smallest = numbers[0]
    biggest = numbers[0]

    # calculations
    for one_number in numbers[1:]:    # start with numbers[1], since biggest/
↪smallest are already numbers[0]
        if one_number < smallest:
            smallest = one_number
        if one_number > biggest:
            biggest = one_number

    # report/return
    return [smallest, biggest]   # returning a list
```

[85]:
```
smallest_and_biggest([10, 5, 3, 17, 12])
```

[85]:
```
[3, 17]
```

[86]:
```
x, y = smallest_and_biggest([10, 5, 3, 17, 12])
```

[87]:
```
x
```

[87]:
```
3
```

[88]:
```
y
```

[88]:
```
17
```

16 Exercise: Evens and odds

1. Write a function, `evens_and_odds`, that returns a dict. The dict contains two keys, `evens` and `odds`. The values are lists, starting off empty.
2. The function should take a list of numbers as an input.
3. Go through each number, and depending on whether it's even or odd, append it to the appropriate list.
4. Return the dict that you have assembled, in which the value for `odds` is a list of odd numbers, and the value for `evens` is a list of even numbers.

Remember: You can check if a number is odd or even by using `n % 2`. If the return value is 1, then it's odd. If it's 0, then it's even.

Example:

```
evens_and_odds([10, 15, 20, 25])  # {'evens':[10, 20], 'odds':[15, 25]}
```

[89]:
```
numbers = [10, 20, 30]
numbers[]
```

```
  Cell In[89], line 2
    numbers[]
            ^
SyntaxError: invalid syntax
```

[90]:
```
def evens_and_odds(numbers):
    output = {'evens': [],
              'odds': []}

    for one_number in numbers:
        if one_number % 2 == 1:   # we got a remainder of 1 when dividing by 2
 ↪-- it's odd!
            output['odds'].append(one_number)  # append one_number to the list
 ↪of odds
        else:
            output['evens'].append(one_number)

    return output
```

[91]:
```
evens_and_odds([10, 15, 20, 25])
```

[91]:
```
{'evens': [10, 20], 'odds': [15, 25]}
```

[ ]:
```
# AB

def evens_and_odds(num):
    evens = []
    odds = []
    for i in num:
        if i % 2 == 0:
            evens.append(i)
        else:
            odds.append(i)

    return {'evens': evens, 'odds': odds}

print(evens_and_odds([24, 15, 23, 56, 99]))
```

[ ]:
```
# PH

def evens_and_odds(number_list):
    return {
        'even': [num for num in number_list if num % 2 == 0],
        'odd': [num for num in number_list if num % 2 != 0]
    }
```

```
print(evens_and_odds([10, 5, 3, 17, 12]))
```

17 Next up

- Default argument values
- Local vs. global variables

18 Default argument values

Normally, when we call a function, the number of arguments we pass must match the number of parameters with which the function was defined. Whether we pass positional or keyword arguments, at the end of the day, Python must have enough values to assign to those parameters.

We know, though, that some functions can be invoked with optional arguments. For example:

```
s = 'a b c d'
s.split(' ')   # pass a space character to split
s.split()      # pass no argument to split at all!
```

This is possible because of default argument values.

When we define a function, we can tell the function that if no argument is passed for a parameter, we have a default value that we want to use.

The way that we indicate this is with `=` after the parameter name, and the value we want it to have.

A function can have as many (or as few) parameters with argument defaults as you want. *HOWEVER*, all mandatory parameters (i.e., without defaults) must come before all optional parameters (i.e., with defaults).

[93]:
```
# define our function, with one parameter that has a default argument value
def hello(name='(no name)'):
    return f'Hello, {name}!'
```

[94]:
```
# parameters: name
# arguments: 'Reuven'

hello('Reuven')
```

[94]:
```
'Hello, Reuven!'
```

[95]:
```
# parameters: name
# arguments: 'world'
hello('world')
```

[95]:
```
'Hello, world!'
```

[96]:
```
# parameters: name
# arguments: '(no name)'
hello()
```

[96]:
```
'Hello, (no name)!'
```

19 Exercise: Count characters

1. Define a function, `count_characters`, that takes two arguments:
   - Mandatory: a string containing text whose characters we want to count
   - Optional: A string whose characters indicate what we want to count. By default, the value will be `'aeiou'`, meaning that we want to count vowels.
2. The function should return a dict whose keys are the characters from the 2nd argument, and whose values are integers – the number of times that each character appeared.

Example:

```
count_characters('hello')            # {'a':0, 'e':1, 'i':0, 'o':1, 'u':0}
count_characters('hello', 'hijkl')   # {'h':1, 'i':0, 'j':0, 'k':0, 'l':2}
```

[97]:
```
def count_characters(s):
    output = {}   # empty dict
    for one_character in 'aeiou':
        output[one_character] = 0

    # count occurrences of looked-for charcters in s
    for one_character in s:
        if one_character in output:     # is this a character we want to count?
            output[one_character] += 1   # add 1 to its count

    return output
```

[98]:
```
count_characters('hello')
```

[98]:
```
{'a': 0, 'e': 1, 'i': 0, 'o': 1, 'u': 0}
```

[99]:
```
def count_characters(s, look_for='aeiou'):
    output = {}   # empty dict
    for one_character in look_for:
        output[one_character] = 0

    # count occurrences of looked-for charcters in s
    for one_character in s:
        if one_character in output:     # is this a character we want to count?
            output[one_character] += 1   # add 1 to its count

    return output
```

[100]:
```
count_characters('hello')
```

[100]:
```
{'a': 0, 'e': 1, 'i': 0, 'o': 1, 'u': 0}
```

[101]:
```
count_characters('hello', 'hijkl')
```

[101]:
```
{'h': 1, 'i': 0, 'j': 0, 'k': 0, 'l': 2}
```

[102]:
```
# AK

def count_characters(s, stype='aeiou'):
    char_dict = {}
    for char in s:
        if char in stype:
            if char in char_dict:
                char_dict[char] += char_dict[char]
            else:
                char_dict[char] = 1
    return char_dict
res = count_characters('hello')
print(res)
res = count_characters('hello', 'hijkl')
print(res)
```

```
{'e': 1, 'o': 1}
{'h': 1, 'l': 2}
```

20 Global vs. local variables

We've seen now that we often want to create variables inside of our functions. These variables are quite useful, letting us accumulate, calculate, etc.

But if I have several functions (as I do!), each of which has a variable called `output`, isn't there a danger that they will affect one another?

No: Any variable we create in a function is a *local* variable. It only exists inside of the function, so long as the function is running. When the function returns, the local variable goes away. This doesn't necessarily mean that the values its variables refer to go away; that depends on whether some other variable is referring to them.

Local variables only exist when the function is running! And they only exist inside of a function.

If you aren't in a function, then you are using *global* variables. Once you set a global variable in Python, it remains assigned until the end of the program.

Why is this a problem? It can get confusing if you have a local variable with the same name as a global variable.

If you're inside of a function, Python first looks for local variables and then (if it can't find one by the name you gave) it looks for global variables.

[104]:
```
x = 100

def myfunc():
    print(f'x = {x}')    # first look for a local x; there isn't any. Now look␣
 ↪for a global x

myfunc()
```

```
x = 100
```

[105]:
```
x = 100  # this creates a global variable x

def myfunc():
    x = 200   # this creates a local variable x
    print(f'x = {x}')

myfunc()
```

```
x = 200
```

[106]:
```
x = 100  # this creates a global variable x

def myfunc():
    x = 200   # this creates a local variable x
    print(f'x = {x}')

print(f'Before, x = {x}')
myfunc()
print(f'After, x = {x}')
```

```
Before, x = 100
x = 200
After, x = 100
```

21 LEGB

This is known as the LEGB rule in Python, and tells you the order in which Python looks for names:

- L – local, inside of a function body
- E – enclosing, if you have a function inside of another functions
- G – global, outside of function body
- B – builtin, names that Python defines and we use without imports (e.g., `len`, `print`, `dict`, `list` all live here)

22 `*args`

We've learned about two types of parameters:

- Mandatory parameters, which take either positional or keyword arguments
- Optional parameters, which takes either positional or keyword arguments, with a default value
- `*args` – a tuple that contains all of the positional arguments that no other parameter was assigned.

This allows us to call a function with an unknown number of positional arguments.

[107]:
```
def myfunc(x, *args):   # "splat args" -- traditional name, because * looks␣
↪like a squashed bug
    return f'x = {x}, args = {args}'
```

[108]:
```
myfunc(10, 20, 30, 40, 50)
```

[108]:
```
'x = 10, args = (20, 30, 40, 50)'
```

[109]:
```
myfunc(10)
```

[109]:
```
'x = 10, args = ()'
```

[110]:
```
myfunc()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[110], line 1
----> 1 myfunc()

TypeError: myfunc() missing 1 required positional argument: 'x'
```

[111]:
```
myfunc(10, args=[10, 20, 30])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[111], line 1
----> 1 myfunc(10, args=[10, 20, 30])

TypeError: myfunc() got an unexpected keyword argument 'args'
```

[112]:
```
# you can call the variable whatever you want; the tradition is to use "args"
# you *MUST* put a * before its name in the parameter list.
```

[113]:
```
# let's take a look at print

help(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.
```

[114]:
```
def myfunc(a, *b):
    return f'a = {a}, b = {b}'
```

[115]:
```
myfunc(10, 20, 30, 40, 50)
```

[115]:
```
'a = 10, b = (20, 30, 40, 50)'
```

[116]:
```
def myfunc(x, *x):
    return f'x = {x}, args = {x}'
```

```
  Cell In[116], line 1
    def myfunc(x, *x):
                   ^
SyntaxError: duplicate argument 'x' in function definition
```

[117]:
```
# if *args is the only parameter, then it gets all of the positional arguments␣
↪as a tuple

def myfunc(*args):
    return f'args = {args}'
```

[119]:
```
myfunc(100, 200, 300)
```

[119]:
```
'args = (100, 200, 300)'
```

23 Exercise: `sum_evens`

1. Write a function, `sum_evens`, that takes any number of numeric arguments.
2. Set up `total` to be 0.
3. The output will be a number, either int or float
4. Have the function iterate over each number in the input
5. If the number is even, add it to `total`.
6. Return `total` to the caller.

Example:

```
sum_evens(10, 11, 12, 13)   # will return 22
```

[120]:
```
def sum_evens(*numbers):
    total = 0
    for one_number in numbers:
        if one_number % 2 == 0:
            total += one_number
    return total
```

[121]:
```
sum_evens(10, 11, 12, 13)
```

[121]: 22

[122]:
```
def sum_evens(numbers):   # what happens if we define the function without *?
    total = 0
    for one_number in numbers:
        if one_number % 2 == 0:
            total += one_number
    return total
```

[123]:
```
sum_evens(10, 11, 12, 13)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[123], line 1
----> 1 sum_evens(10, 11, 12, 13)

TypeError: sum_evens() takes 1 positional argument but 4 were given
```

24 Next week: Modules and packages

[ ]:
```

```