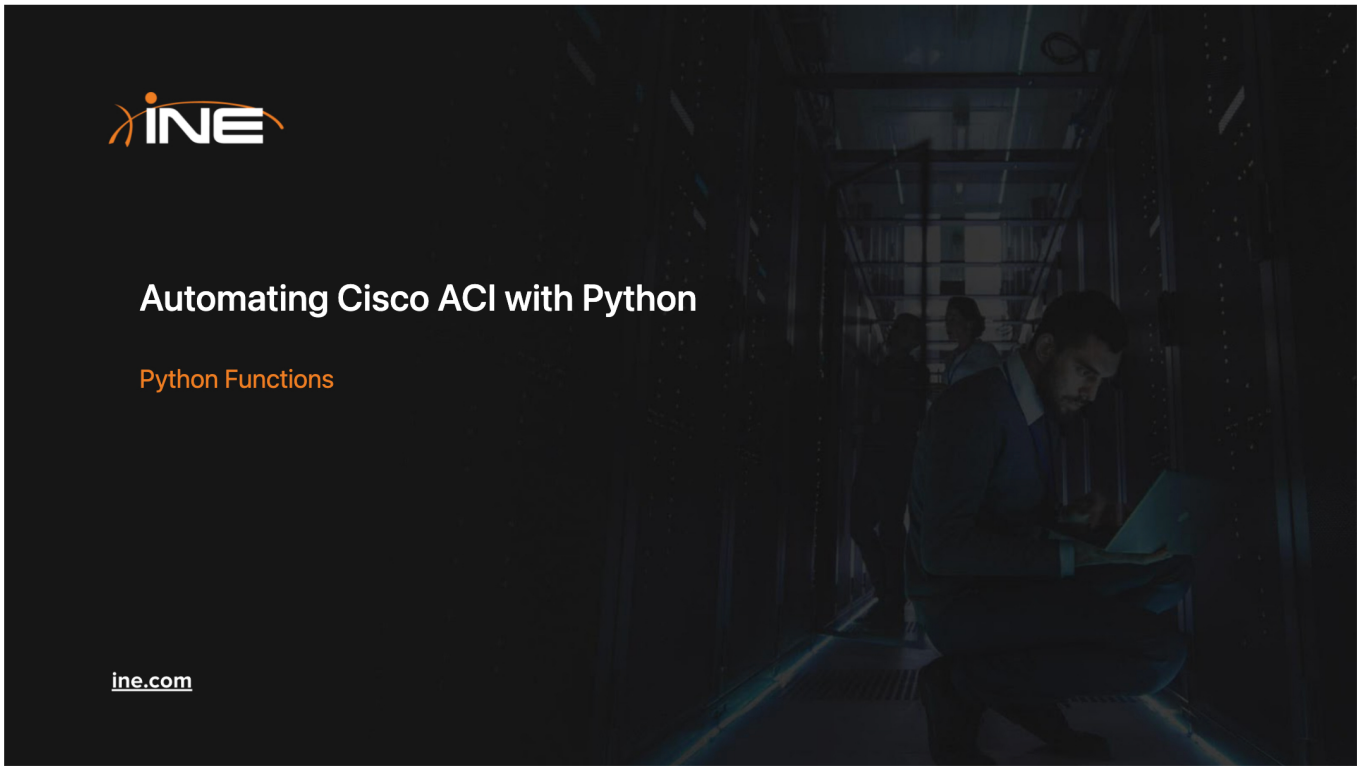

We can define a function as a group of statements with a name which are executed when it is called.

Functions can receive data, known as parameters. Also, they can return values as a result.

Let's dig a little bit deeper. Functions in Python are very intuitive. Let's start with an example of a function without parameters:

In [37]:
```
def hello():
    return "Hello World"
```

The `def` keyword indicate the *definition* of a function, followed by a name and a list of arguments (which this function doesn't receive). The `return` statement is used to break the flow of the function and return a value back to the caller:

In [38]:
```
result = hello()
```

In [39]:
```
result
```

Out[39]:
```
'Hello World'
```

If a function doesn't explicitly include a `return` statement, Python will return `None` by default:

In [40]:
```
def empty():
    x = 3
```

In [41]:
```
result = empty()
```

In [42]:
```
print(result)
```

```
None
```

In [16]:
```
def say_hi():
    return "Hi"
```

In [18]:
```
result = say_hi()
```

In [19]:
```
print(result)
```

```
Hi
```

Function parameters

There's a lot that can be done with Python parameters; including default and named parameters, and even variable/dynamic ones. But for now, we'll just focus on the basics. Function parameters are listed at the function definition, and they're part of the function's local scope:

In [43]:
```
def add(x, y):
    return x + y
```

In [44]:
```
add(2, 3)
```

Out[44]:
```
5
```

We can set predefined values to our parameters to use when the user doesn't give a value to that parameter:

In [6]:
```
def add(x, y=10):
    return x + y
```

In [7]:
```
add(2)
```

Out[7]:
```
12
```

In [8]:
```
add(2, 5)
```

Out[8]:
```
7
```

Also we can use the parameter name to pass the parameters in a custom order:

In [12]:
```
add(x=3)
```

Out[12]:
```
13
```

In [13]:
```
add(x=3, y=5)
```

Out[13]:
```
8
```

In [14]:
```
add(y=5, x=3)
```

Out[14]:
```
8
```

But we can't use named parameters missing a previous parameter, in that case we'll get an error:

In [15]:
```
add(y=5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-e4ea2dc698a0> in <module>
----> 1 add(y=5)

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

We can also define functions that accept variable number of positional arguments, using the star args `*`:

In [44]:
```
def add(*args):
    print(args)
    return sum(args)
```

In [45]:
```
add(1, 1, 1)
```

```
(1, 1, 1)
```

Out[45]:
```
3
```

In [46]:
```
add(1)
```

```
(1,)
```

Out[46]:
```
1
```

Or use ****kwargs** if we want to accept variable number of named arguments:

In [47]:
```
def add(**kwargs):
    print(kwargs)
    return sum(kwargs.values())
```

In [48]:
```
add(a=1, b=2, z=5, d=10)
```

```
{'a': 1, 'b': 2, 'z': 5, 'd': 10}
```

Out[48]:
```
18
```

Or both by position and name:

In [51]:
```
def add(*args, **kwargs):
    print(f"args {args}")
    print(f"kwargs {kwargs}")
    return sum(kwargs.values())
```

In [52]:
```
add()
```

```
args ()
kwargs {}
```

Out[52]:
```
0
```

In [55]:
```
add(4, 7, 2)
```

```
args (4, 7, 2)
kwargs {}
```

Out[55]:
```
0
```

In [56]:
```
add(a=10, b=5)
```

```
args ()
kwargs {'a': 10, 'b': 5}
```

Out[56]:
```
15
```

In [57]:
```
add(4, 7, 2, a=10, b=5)
```

```
args (4, 7, 2)
kwargs {'a': 10, 'b': 5}
```

Out[57]:
```
15
```

In [53]:
```
add(a=10, b=5, 2)
```

```
  File "<ipython-input-53-1a14e75a2377>", line 1
    add(a=10, b=5, 2)
                   ^
SyntaxError: positional argument follows keyword argument
```

In [97]:
```
def first_items(mylist, max=4):
    i = 0
    while i < max:
        item = mylist[i]
        print("Index", i, "Value", item)
        i = i+1
        if type(item) == str:
            print("The length of the string is", len(x))
        print('---')
```

In [80]:
```
nums = [5, 9, 2.134, 99, 33, 44, 55, 983, 45]
```

In [81]:
```
first_items(nums)
```

```
Index 0 Value 5
---
Index 1 Value 9
---
Index 2 Value 2.134
---
Index 3 Value 99
---
```

In [79]:
```
first_items(nums, 7)
```

```
Index 0 Value 5
---
Index 1 Value 9
---
Index 2 Value 2.134
---
Index 3 Value 99
---
Index 4 Value 33
---
Index 5 Value 44
---
Index 6 Value 55
---
```

In [82]:
```
import math

def distance(a, b, c=0, d=0, e=0):
    hypot = math.sqrt(a**2 + b**2 + c**2 + d**2 + e**2)
    return hypot
```

In [83]:
```
distance(10, 20)
```

Out[83]:
```
22.360679774997898
```

In [84]:
```
distance(10, 20, 30)
```

Out[84]:
```
37.416573867739416
```

Local vs. Global scope

In [28]:
```
value_1 = 5

def print_value():
    print(f'Inside: {value_1}')
    return value_1
print_value()

print(f'Outside: {value_1}')
```

```
Inside: 5
Outside: 5
```

In [29]:
```
def print_value():
    value_2 = 7
    print(f'Inside: {value_2}')
    return value_2
print_value()

print(f'Outside: {value_2}')
```

```
Inside: 7
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-29-146a85d947a2> in <module>
      5 print_value()
      6
----> 7 print(f'Outside: {value_2}')

NameError: name 'value_2' is not defined
```

In [30]:
```
def print_value():
    global value_3
    value_3 = 8
    print(f'Inside: {value_3}')
    return value_3
print_value()

print(f'Outside: {value_3}')
```

```
Inside: 8
Outside: 8
```

Return values

In [58]:
```
def get_list_length(l):
    return len(l)
```

In [62]:
```
list_1 = [1, 5, 2, 6, 9]
list_2 = [2, 6, 9]
```

In [65]:
```
result = get_list_length(list_1)

print(result)
```

```
5
```

In [66]:
```
result = get_list_length(list_2)

print(result)
```

```
3
```

Returning multiple values

In [73]:
```
def get_list_length_and_min_value(l):
    return len(l), min(l)
```

In [74]:
```
result = get_list_length_and_min_value(list_1)

print(result)
```

```
(5, 1)
```

In [75]:
```
l_length, l_min_value = get_list_length_and_min_value(list_1)
```

In [76]:
```
print(l_length)
```

```
5
```

In [77]:
```
print(l_min_value)
```

```
1
```

In [88]:
```
def foo_function(a, b, c=None):
    if c is None:
        return(["No c", a, b])
    else:
        return [c, a, b]
```

In [90]:
```
result = foo_function(88, 77)

print(result)
```

```
['No c', 88, 77]
```

In [91]:
```
result = foo_function(88, 77, 66)

print(result)
```

```
[66, 88, 77]
```

Comments and docstrings

In [98]:
```
help(first_items)
```

```
Help on function first_items in module __main__:

first_items(mylist, max=4)
```

In [99]:
```
def first_items(mylist, max=4):
    "Print off max elements from a list"
    i = 0
    while i < max:
        item = mylist[i]
        print("Index", i, "Value", item)
        i = i+1
        # Here we only want more info about strings
        if type(item) == str:
            print("The length of the string is", len(x))
        print('---')   # A visual divider is helpful
```

In [100…
```
help(first_items)
```

```
Help on function first_items in module __main__:

first_items(mylist, max=4)
    Print off max elements from a list
```

In [101…
```
help(math.sqrt)
```

```
Help on built-in function sqrt in module math:

sqrt(x, /)
    Return the square root of x.
```

In [95]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

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

Exercises

Exercise 1

Make a function using the following formula:

$$f(x, y) = 2x + y$$

In [3]:
```
# your code goes here
```

Exercise 2

Make a function that receives a string as parameter and return the last letter on it.

In [ ]:
```
# your code goes here
```

In [1]:
```
# Solution

def get_last_letter(string):
    return string[-1]
```

Exercise 3

Make a function that receives a list as argument and returns how many elements it has; if it doesn't have any element return an error message.

In [ ]:
```
# your code goes here
```

In [2]:
```
# Solution

def get_list_info(l):
    if len(l) == 0:
        return "Error"
    else:
        return len(l)

print(get_list_info([]))
print(get_list_info([1, 2, 3]))
```

```
Error
3
```

Exercise 4

We're going to explore how to receive parameters and operate with them in your functions. In the editor you'll see a simple function that accepts just one parameter (named `x` in this case). This function's job is to multiply `x` by 2 and return the result. For example, if we pass 3 as the parameter, we're going to receive `3 * 2` as the result:

```
multiply_by_two(3)  # 6
```

Parameters are matched by position. In this case `multiply_by_two` receives only one parameter (`x`), so when we invoke it passing the number `3`, it gets assigned to `x`.

Now go ahead and try implementing the `multiply_by_two` function by yourself.

In [ ]:
```
def multiply_by_two(x):
    # your code goes here
    pass

two_times_two = multiply_by_two(2)
three_times_two = multiply_by_two(3)
five_times_two = multiply_by_two(5)

print("2x2 = {}".format(two_times_two))
print("3x2 = {}".format(three_times_two))
print("5x2 = {}".format(five_times_two))
```

In [ ]:
```
# Solution

def multiply_by_two(x):
    return x * 2

two_times_two = multiply_by_two(2)
three_times_two = multiply_by_two(3)
five_times_two = multiply_by_two(5)

print("2x2 = {}".format(two_times_two))
print("3x2 = {}".format(three_times_two))
print("5x2 = {}".format(five_times_two))
```

Exercise 5

Define a function `conditional_multiplication` that receives a boolean and a number and returns a number. If the boolean is True the number returned is multiplied by 10. In other case, it returns it unchanged.

Examples:

```
# Boolean is True, multiply it by 10
>>> conditional_multiplication(True, 2)
20

# Boolean is False, return it as it is
>>> conditional_multiplication(False, 5)
5
```

In [ ]:
```
def conditional_multiplication(a_condition, number):
    # your code goes here
    pass
```

In [ ]:
```
# Solution

def conditional_multiplication(a_condition, number):
    if a_condition:
        number *= 10
    return number
```

Exercise 6

Define a function `traffic_light` that receives a color and returns:

- 'stop' if the color is `red`
- 'slow down' if the color is `yellow`
- 'go' if the color is `green`

Examples:

```
>>> traffic_light('red')
'stop'
>>> traffic_light('yellow')
'slow down'
>>> traffic_light('green')
'go'
```

In [ ]:
```
def traffic_light(color):
    # your code goes here
    pass
```

In [ ]:
```
# Solution

def traffic_light(color):
    if color == 'red':
        return 'stop'
    elif color == 'yellow':
        return 'slow down'
    elif color == 'green':
        return 'go'
```

Exercise 7

Define a function `get_grade_letter` that receives a score and you should return:

- 'A' if the score is 90 or above
- 'B' if the score is 80 to 89
- 'C' if the score is 70 to 79
- 'D' if the score is 60 to 69
- 'F' if the score is less than 60

Examples:

```
>>> get_grade_letter(93)
'A'
>>> get_grade_letter(80)
'B'
>>> get_grade_letter(75)
'C'
>>> get_grade_letter(67)
'D'
>>> get_grade_letter(42)
'F'
```

In [ ]:
```
def get_grade_letter(score):
    # your code goes here
    pass
```

In [3]:
```
# Solution

def get_grade_letter(score):
    if score >= 90:
        return 'A'
    elif score >= 80:
        return 'B'
    elif score >= 70:
        return 'C'
    elif score >= 60:
        return 'D'
    else:
        return 'F'
```

Exercise 8

Complete the function `double_down` so that it receives a number `num` and an integer `times_doubled` and returns `num` doubled `times_doubled` amount of times. Use a while loop to do this (and for everyone's sake, no infinite loops!).

Examples:

```
>>>  double_down(3, 3)
24

>>>  double_down(2, 0)
2

>>>  double_down(4, 1)
8
```

In [ ]:
```
def double_down(num, times_doubled):
    # your code goes here
    pass
```

In [ ]:
```
# Solution

def double_down(num, times_doubled):
    count = 0
    while count < times_doubled:
        num *= 2
        count += 1
    return num
```

Exercise 9

Use a while loop to complete the function `feel_the_power` so that it takes the `initial_number` and multiplies it by itself power number of times.

Examples:

```
>>> feel_the_power(3, 0)
1

>>> feel_the_power(3, 1)
3

>>> feel_the_power(3, 2)
9

>>> feel_the_power(3, 3)
27
```

> Hint: You'll need additional variables to keep track of both the result and how many times you've multiplied it by itself.
>
> Hint 2: Remember, with multiplication, you start your result at 1 instead of 0 because if you multiply by 0 you get 0.
>
> Hint 3: NO INFINITE LOOPS!

In [ ]:
```
def feel_the_power(initial_number, power):
    # your code goes here
    pass
```

In [4]:
```
# Solution

def feel_the_power(initial_number, power):
    result = 1
    power_count = 0
    while power_count < power:
        result *= initial_number
        power_count += 1
    return result
```

Exercise 10

Today we're going to write a function to convert what a pirate would say into the way a dog would say it. Roo roo roo roo!

Complete the function `convert_pirate_to_dog` with a for loop so that it goes through each letter of `pirate_saying` and then adds 'oo' after each letter.

Remember that strings are immutable, so you will have to create a new string for the result of how a dog would sound.

Examples:

```
>>>  convert_pirate_to_dog("arrrr")
'aoorooroorooroo'
```

> Hint: To add letters to the result string, use the += operator.

In [ ]:
```
def convert_pirate_to_dog(pirate_saying):
    # your code goes here
    pass
```

In [ ]:
```
# Solution

def convert_pirate_to_dog(pirate_saying):
    result_in_dog_speak = ""
    for letter in pirate_saying:
        result_in_dog_speak += letter
        result_in_dog_speak += "oo"
    return result_in_dog_speak
```

Exercise 11

Write a function `is_even_and_contains_red` that receives a list (containing colors) and returns True if the list contains the color `"red"` **AND** has an even number of elements. False, otherwise.

Check the examples:

```
# 4 elements (even) with red:
is_even_and_contains_red(['red', 'blue', 'green', 'white'])  # True

# 3 elements (odd!) with red:
is_even_and_contains_red(['red', 'blue', 'green'])  # False

# 2 elements (even!) **WITHOUT** red:
is_even_and_contains_red(['white', 'blue', 'green', 'black'])  # False

# 3 elements (odd!) **WITHOUT** red:
is_even_and_contains_red(['white', 'blue', 'green'])  # False
```

In [102…

```
def is_even_and_contains_red(a_list):
    # your code goes here
    pass
```

In [ ]:

```
# Solution

def is_even_and_contains_red(a_list):
    if len(a_list) % 2 == 0 and 'red' in a_list:
        return True
    return False
```