

### Return Statements:

- Exist only inside functions
- Mark the end of the algorithm in the function body
  - We say we “exit” a function when we hit a return statement
- Provide the value that the function outputs when we call it
- Functions implicitly return None if an explicit return statement is not provided

Print vs. Return:

Print	Return
Can be used anywhere	Can only be used inside a function
Used to display values in interpreter	Used to provide values for future computations

Example: suppose we had the following function

```
def say_hi(name):  
    greeting = "Hi, " + name  
    print(greeting)  
  
phrase = say_hi("Carnegie")
```

The value of the variable phrase would be None because we did not return anything in say\_hi.

### Optional Parameters:

You can define functions with optional parameters in addition to (or instead of) required parameters. Optional parameters are defined by setting the parameter equal to a default value in the function definition. For example, suppose we have the function

```
def say_hi(name, title=""):  
    greeting = "Hi, " + title + " " + name  
    return greeting
```

Here, **name** is a required parameter and **title** is an optional parameter. The default value of **title** is the empty string. When calling this function, we can pass in either one or two values.

```
print(say_hi("Carnegie", "Mr. "))    # prints Hi, Mr. Carnegie  
print(say_hi("Carnegie"))            # prints Hi, Carnegie
```

In the first print statement, we gave it an argument for **title**, which was used in the function as normal. In the second print statement, we did not provide an argument for **title**, so the function used the default value we specified.

You can have multiple optional parameters:

```
def say_bye(name, title="", goodbye="Bye"):
    farewell = goodbye + ", " + title + " " + name
    return farewell

print(say_bye("M")) # Bye, M
print(say_bye("M", "Ms. ")) # Bye, Ms. M
print(say_bye("M", "Ms.", "Adios")) # Adios, Ms. M
print(say_bye("M", goodbye="Farewell")) # Farewell, M
print(say_bye("M", goodbye="Adieu", title="Ms. ")) # Adieu, Ms. M
```

This function has two optional parameters: **title**, which is defaulted to the empty string and **goodbye**, which is defaulted to the string Bye. When calling the function, you can pass in the required parameter, **name**, and any combination of the optional parameters.

When passing in optional parameters out of order (as shown in the last two print statements), you need to specify which parameter you're providing the value for.

### **Library Functions:**

Python has access to a large number of libraries, where built-in functions are stored. To use one of these functions, you need to import the library first. It is most common to have these import statements at the top of your file so that all functions within the file have access to the library.

There are a couple different ways to import a library. The way you import a library then determines how you can call a function within that library.

Import statement	Function call
<code>import library_name</code>	<code>library_name.function_name(params)</code>
<code>from library_name import function_name</code>	<code>function_name(params)</code>

As a more concrete example, suppose we want to use the **randint** function located in the **random** library. The **randint** function takes in two parameters a and b, and returns a random integer in the range [a, b] (inclusive on both ends).

Method #1 of importing:

```
import random
print(random.randint(1, 10)) # prints random integer between 1 and 10
```

Method #2 of importing:

```
from random import randint
print(randint(1, 10))          # prints random integer between 1 and 10
```

### **Variable Scope:**

Variable scope refers to where variables can be accessed. There are two types of scope: **local** and **global**.

A variable has **local** scope if it is defined inside a function. (This includes the parameters of a function!) This means that the variable is accessible only within this function. If you try to access it outside of the function or inside a different function, you will get a `NameError`.

A variable has **global** scope if it is defined outside of any function. This means that the variable is accessible anywhere, both inside of functions and outside.

When Python looks for a variable, it will first check the local scope and then the global scope. This means that if there is a local variable that has the same name as a global variable, Python will use the local one in calculations unless told otherwise. For example:

```
greeting = "Hello"                # global variable greeting
def say_hi(name):
    greeting = "Hi"                # local variable greeting
    message = greeting + ", " + name
    return message

print(say_hi("Carnegie"))          # prints Hi, Carnegie
```

The print statement defaults to using the local variable greeting when called.

Local and global variables with the same name are completely separate. Changing the local variable inside a function will not change the global variable, and vice versa. To change the global variable inside of a function, use the **global** keyword. For example:

```
greeting = "Hello"                # global variable greeting
def say_hi(name):
    global greeting                # tell Python to use global variable
    greeting = "Hi"                # updates the global variable
    message = greeting + ", " + name
    return message

print(greeting)                   # changed to "Hi"
```

Each function has its own local scope. This means that you can have multiple functions with a variable of the same name in each of them. These variables are local to their specific function and do not have anything to do with each other. For example:

```
greeting = "Hello"                                # global variable greeting

def arrive(name):                                  # name local to arrive function
    message = greeting + ", " + name               # uses global greeting
    name = "Carnegie"                             # updates local name
    return message

def leave(name):                                   # name local to leave function
    greeting = "Goodbye"                          # local variable greeting
    message = greeting + ", " + name               # uses local greeting
    return message

name = "Andrew"                                   # global variable name
print(arrive(name))                               # Hello, Andrew
print(leave(name))                               # Goodbye, Andrew
```

In general, it is recommended to avoid global variables as they can cause hard-to-debug issues.

### **Booleans:**

Booleans (bools) are a new data type that can only be either True or False (capitalization required).

Operation	Result
2 < 4	True
4.5 >= 10	False
4 <= (2+2)	True
(2+4) == 6	True
9 != (10-1)	False
"Hi" != "Hello"	True (can check for equality between strings)
"Hello" == 42	False (can check for equality between different types)
4 < "ten"	Error (cannot use other operators on different types)

## Logical Operators:

We use logical operators to combine booleans together. The three main ones are:

1. **and**: takes two booleans and evaluates to True if **both** booleans are True  
useful for cases where multiple conditions all need to be met at the same time
2. **or**: takes two booleans and evaluates to True if **at least one** of the booleans are True  
useful for cases where there are multiple valid conditions to choose from
3. **not**: takes one boolean and flips the value  
useful for cases where it's easier to express a condition by using the converse

Example: suppose we set `x = 10`

```
((x > 5) or (x**2 > 50)) and (x == 20)
=> ((10 > 5) or (10**2 > 50)) and (10 == 20) # plug in value for x
=> (True or True) and (False)                # evaluate each set of ()
=> True and False
=> False
```

Python uses short-circuit evaluation, which means that it only evaluates the second half of the operation if it has to. It can skip the second half entirely in two cases:

1. False **and** \_\_\_\_ → regardless of what the second argument is, this will always be False
2. True **or** \_\_\_\_ → regardless of what the second argument is, this will always be True

The second argument can even be something that would ordinarily crash the program. Since Python ignores the argument completely in these cases, the program will continue running just fine.

## Conditionals:

Conditionals are an important control structure that allow us to run different parts of the code in different circumstances.

The most basic form is just one **if statement**, that looks like this:

```
if (some boolean expression):
    loop body
```

The expression after the `if` keyword must evaluate to either True or False. The loop body will only execute if the expression is True. Otherwise, it will skip over the loop body and continue with whatever code follows the `if` statement. Similar to functions, the entire loop body is indented underneath the `if` statement. Anything that is not indented is not part of the `if` statement.

We can extend this if statement to include an **else case**. This is useful if we have two mutually exclusive options and need to choose one or the other. The syntax looks like this:

```
if (some boolean expression):
    loop body 1
else:
    loop body 2

{rest of code}
```

The program will check and see if the boolean expression is True. If it is, it will evaluate loop body 1 and then proceed to the rest of the code (skipping loop body 2). If the boolean expression is False, it will skip the if case and evaluate loop body 2 before continuing to the rest of the code. As a more concrete example, suppose we had

```
x = 7

if x < 10:
    print("small")
else:
    print("big")
```

This will print “small” since  $7 < 10$  so the if case is True. If we had  $x = 15$ , then this code would print “big” since the if case would be False and so it would evaluate the else case.

As mentioned before, if/else statements are for mutually exclusive cases – i.e. in any given run, only the if or else case will be evaluated – both cannot be run at the same time.

The last extension of this control structure is one that allows us to choose between three or more options. This takes the form of an **elif case** that goes between the **if** and **else** cases like so:

```
x = 23

if x < 10:
    print("small")
elif x < 100:
    print("medium")
else:
    print("big")
```

This code would print “medium” because the first case is False (since 23 is not  $< 10$ ) and the second case is True. As with the previous example, if/elif/else cases are mutually exclusive so once the program evaluates one of them, it will skip the rest.

Note that these cases are evaluated sequentially. If we had  $x = 7$ , then the code will only print “small” because it will see that the first case is True and thus will evaluate the code in that first if block and then skip the rest.

In a conditional statement, you must have at least an **if** statement. Then you can have as many **elif** cases as you want and either 0 or 1 **else** case. Note that you cannot have unindented code in between the cases. So, for example, you can't do something like this:

```
if x < 10:
    print("small")
    print("here")
elif x < 100:
    print("medium")
else:
    print("big")
```

This is because that **print("here")** statement will break up the chain of if/elif/else cases, which always have to be connected one after the other.

We can put if statements inside functions. It is often the case that we use if statements to make our function return different things in different cases. For example, suppose we wanted to write a function that calculates the average given a total and number of elements n. Mathematically, this is pretty simple, we just need to divide the total by n. But as programmers, we want to make sure our function accommodates edge cases. For example, what if the n that gets passed in is 0 or negative? In either case, we shouldn't be able to calculate a meaningful average and we want our program to reflect that. We can use conditionals to do this:

```
def find_average(total, n):
    if n <= 0:
        return "n/a"
    else:
        return total / n
```

This code will check if the input n was 0 or negative and return "n/a" if it is. Otherwise, it will calculate the average as expected.

Note that it's okay that we have two return statements here. Since the **if/else** cases are mutually exclusive, our program will only ever hit one return statement. Remember that return statements always exit the function immediately so if we had additional code after this if/else block, it would never be executed because we hit a return statement in both cases.

Syntactically, we can also write this function like this:

```
def find_average(total, n):
    if n <= 0:
        return "n/a"
    return total / n
```

The performance will be exactly the same because if the first case is True, then we will hit the **return "n/a"** line and exit the function immediately, which means we'll ignore the **return total/n** line. If the first case is not True, then we'll skip the body of the if statement, which is the first return statement and continue on to the rest of the code, which is the second return statement and so we'll return **total/n** anyway as needed. This is just a way to shorten the function slightly –

it's not necessary to write code like this. It also only works with return statements because of the aforementioned idea that return statements exit the function immediately. (Don't worry if this is confusing to you right now – once you're more comfortable with Python, it may come more naturally :))

We can also nest conditionals within each other like this:

```
if first == True:
    if second == True:
        print("both true!")
    else:
        print("second not true")
else:
    print("first not true")
```

In this example, we have two sets of conditionals (highlighted red and blue for visual clarity). The indentation levels also help you see which sets of conditionals “match”.

Sometimes, we can condense nested conditionals by using logical operators (and/or/not). For example, if we had:

```
if first == True:
    if second == True:
        print("both true!")
```

This can be rewritten as:

```
if (first == True) and (second == True):
    print("both true!")
```

Where we've linked the two conditions together using the logical operator **and**. Note that syntax-wise, if you have logical operators in an if statement, it's usually best to put each component in parentheses as we have above. Otherwise, Python may misinterpret the statement in strange ways and make it difficult to debug.

Let's look at a concrete example. Suppose we want to write a function that determines whether someone can rent a car. The two requirements for being able to rent a car are 1) they are at least 26 and 2) they have a valid driver's license. We'll pass in the variables **age** and **has\_license** into our function, where **age** is an integer and **has\_license** is a boolean.

```
def can_rent_car(age, has_license):
    if age >= 26:
        if has_license == True:
            return True
        else:
            return False
    else:
        return False
```



This is a completely valid way to write this function and logically, it's probably one of the clearest ways. However, there are a few ways we can shorten it so that it's fewer lines and generally looks neater. The first thing we can do is to link together the two if statements using **and**, similar to the example we wrote above:

```
def can_rent_car(age, has_license):
    if (age >= 26) and (has_license == True):
        return True
    else:
        return False
```

The second way we can shorten this function is that we actually don't need the "**== True**" part when checking **has\_license**. We can actually write it like this:

```
def can_rent_car(age, has_license):
    if (age >= 26) and (has_license):
        return True
    else:
        return False
```

This is because when we use logical operators, we need each component to evaluate to a boolean. **has\_license** is already a boolean so we can just use it directly as part of this conditional.

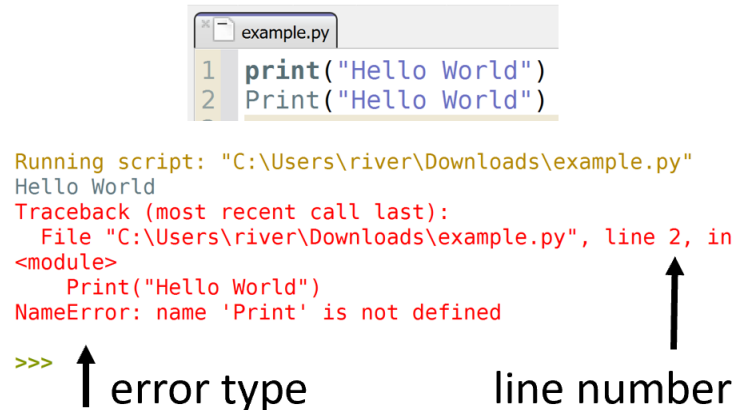
If we wanted to make this function even shorter, we could write it like this:

```
def can_rent_car(age, has_license):
    return (age >= 26) and (has_license)
```

This jump might be a bit confusing. In the previous version, we cased on the two conditions and returned True or False accordingly. However, notice that the boolean that we return is the same as what the conditional evaluates to. In other words, we return True if both conditions are True and False if either one (or both) are False. So, we can just return the result of these conditions directly since it will evaluate to the correct boolean in all cases. (Again, don't worry if this last part was confusing – it will hopefully come more naturally as you become more familiar with Python!)

## Debugging:

Previously, we discussed three types of errors you'll encounter when coding: syntax, runtime, and logical errors. Syntax and runtime errors are easy to detect because Python will tell you directly that they exist. Whenever you get an error message, look for the line number and the error type. For example, a runtime error might give you a message like this:

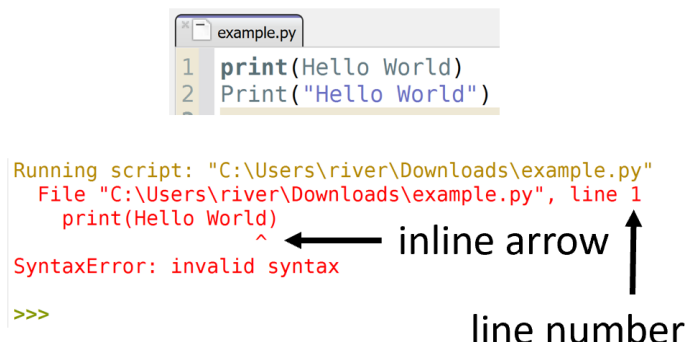


The image shows a code editor window titled 'example.py' with two lines of code: `1 print("Hello World")` and `2 Print("Hello World")`. Below the editor, the terminal output shows the script running successfully on line 1, printing 'Hello World', and then encountering a `NameError` on line 2 because the variable `Print` is not defined. An arrow points from the text 'error type' to the `NameError` message, and another arrow points from the text 'line number' to the `line 2` in the error message.

```
Running script: "C:\Users\river\Downloads\example.py"
Hello World
Traceback (most recent call last):
  File "C:\Users\river\Downloads\example.py", line 2, in
    <module>
      Print("Hello World")
NameError: name 'Print' is not defined
>>>
```

It's telling us that the error occurred on line 2 of our file and that it's a `NameError`. The line number will indicate where to look for the issue and the type of error should give you more information about how to fix the issue. In this case, we got an error because we used a capital p in our `Print` statement and Python doesn't recognize `Print` as a command.

For syntax errors, Python tries to be even more helpful and tell us where in the line it found the error. For example, we might see a message like this:



The image shows a code editor window titled 'example.py' with two lines of code: `1 print(Hello World)` and `2 Print("Hello World")`. Below the editor, the terminal output shows a `SyntaxError` on line 1 because `print(Hello World)` is not a valid Python statement. An arrow points from the text 'inline arrow' to the caret (^) symbol above the closing parenthesis in the error message, and another arrow points from the text 'line number' to the `line 1` in the error message.

```
Running script: "C:\Users\river\Downloads\example.py"
File "C:\Users\river\Downloads\example.py", line 1
  print(Hello World)
        ^
SyntaxError: invalid syntax
>>>
```

In this example, it tells us that the error occurs on line 1, specifically at the end. The error type is also marked as `"SyntaxError"`. In this case, the issue is that we didn't use quotes when printing out `"Hello World"` so Python was unable to evaluate it as a string. Note that you should take the inline arrow with a grain of salt – sometimes it may not be super accurate, but it's always a good starting point. If the inline arrow is at the beginning of the line, there's also a chance that the issue is occurring in the line above the indicated line (for example, mismatched parentheses in the previous line is a very common issue I've seen).

Debugging logical errors is a bit trickier. These errors are when your code doesn't perform the way you want it to. It may run without any issues but still produce the wrong answer, and Python can't flag this for you. To catch logical errors, we write test cases using **assert statements**. These statements take in an expression that evaluates to a boolean and if the boolean is False, will throw an `AssertionError` that halts the program (similar to what a syntax or runtime error would do). If the boolean is True, it will not do anything.

To illustrate how assert statements work, let's look at an example. Suppose we're trying to write a function **find\_average(total, n)** that calculates the average given a total and number of elements. We implemented the function like this:

```
def find_average(total, n):  
    if n <= 0:  
        return "Cannot compute the average"  
    return total // n
```

To test this, we want to call this function on several different inputs and check that it outputs the correct value. So, we would write an assert statement like this:

```
assert(find_average(20, 4) == 5)
```

We call the built-in **assert** function (which is like calling the **print** statement) and give it a boolean expression. In this case, the boolean expression comes from calling the **find\_average** function with inputs 20 and 4 and checking that the output is equal to 5. If our implementation of **find\_average** doesn't return 5 when given 20 and 4 as arguments, then this assertion will fail.

To be robust, we always want to write multiple test cases that comprehensively cover all cases. So for example, I would also want a test that finds the average when the answer is not a whole number. I'd also want a test for when the passed in **n** is negative or zero to check that our function handles that case correctly.

```
assert(find_average(13, 2) == 6.5)  
assert(find_average(10, 0) == "Cannot compute the average")
```

When I run my file with these cases, I'll get an error like

```
Running script: "C:\Users\river\Downloads\example.py"  
Traceback (most recent call last):  
  File "C:\Users\river\Downloads\example.py", line 13, in <module>  
    assert(find_average(13, 2) == 6.5)  
AssertionError
```

This indicates that the error was thrown on the test case **find\_average(13, 2)**, where the expected output was 6.5 but our function apparently produced something different. To debug this, the first step is always to figure out what your function is returning and compare it to what it should be. We can do this by printing out this function call in our file:

```
print(find_average(13, 2))
```

This print statement needs to be before the test case is run (I would recommend commenting out the test case until you're ready to test again, just to prevent your console from being cluttered up by error messages). When we run this, we'll see that this will print out 6. We know the expected output is 6.5, so it seems like our function is cutting off the decimal point. If we go back and look at our implementation, we can see that we're using integer divide to calculate the average when we should be using normal division. If we fix that, our function will pass.

Typically, debugging logical error will be a more involved process, particularly when your functions contain more than 2-3 lines. There are some strategies to help you debug:

1. **Rubber duck debugging:** this is the process of explaining your code out loud to someone (friend, classmate, unwilling family member, etc.) or something (e.g. rubber duck). The idea is that when talking your thought process through out loud, you may discover a missing piece of logic or mistake in your code. It's not the same as going through it mentally – it really does have to be spoken.
2. **Print and experiment:** this approach is particularly useful when you're using lots of variables. Chances are one of the variables is being set incorrectly at some point but it could be difficult/impossible to tell which one it is by just staring at your code. The idea of this approach is to put print statements in select places throughout the code to narrow down where the issue is occurring. An important caveat of this approach is that these print statements have to be **intentional**. It won't do you any good to just chuck a bunch of print statements in there willy-nilly – that will probably make it even more confusing.

For each print statement you put in, you should know in your mind (or on a piece of paper) the values that you expect to be printed out. Then when you actually run the code with your print statements, compare what's printed out to what you expect. If they all match, put in a new set of print statements. If one doesn't match, then that's part of the issue and should be a starting point to debugging why they don't match.

As a helpful tip, when printing out variables, I like to label them in the print statement. For example, if I had the variable **a**, then my print statement would be something like:

```
print("a: ", a)
```

This way, when the values come out in the console, I know exactly what value **a** corresponds to.

3. **Trace your code:** this is the most manual approach. Essentially, it involves you tracing through your code by hand step-by-step. At each line, you manually calculate the values your code is producing by literally evaluating the code with your brain instead of the computer. Then compare the value the code is producing with what you expect it to (i.e. when you wrote the code, what value did you expect it to have at this point). This approach is similar to the previous one except that you're doing both sides by hand instead of having the computer print one side.