codecademy

Syntax

Comments

A comment is a piece of text within a program that is not executed. It can be used to provide additional information to aid in understanding the code.

The `#` character is used to start a comment and it continues until the end of the line.

```
# Comment on a single line

user = "JDoe" # Comment after code
```

`print()` Function

The `print()` function is used to output text, numbers, or other printable information to the console. It takes one or more arguments and will output each of the arguments to the console separated by a space. If no arguments are provided, the `print()` function will output a blank line.

```
print("Hello World!")

print(100)

pi = 3.14159
print(pi)
```

Strings

A string is a sequence of characters (letters, numbers, whitespace or punctuation) enclosed by quotation marks. It can be enclosed using either the double quotation mark `"` or the single quotation mark `'`.

If a string has to be broken into multiple lines, the backslash character `\` can be used to indicate that the string continues on the next line.

```
user = "User Full Name"
game = 'Monopoly'

longer = "This string is broken up \
over multiple lines"
```

Variables

A variable is used to store data that will be used by the program. This data can be a number, a string, a Boolean, a list or some other data type. Every variable has a name which can consist of letters, numbers, and the underscore character `_`.

The equal sign `=` is used to assign a value to a variable. After the initial assignment is made, the value of a variable can be updated to new values as needed.

```
# These are all valid variable names and
assignment

user_name = "@sonnynomnom"
user_id = 100
verified = False

# A variable's value can be changed
after assignment

points = 100
points = 120
```

Errors

The Python interpreter will report errors present in your code. For most error cases, the interpreter will display the line of code where the error was detected and place a caret character `^` under the portion of the code where the error was detected.

```
if False ISNOTEQUAL True:
                 ^
SyntaxError: invalid syntax
```

SyntaxError

A SyntaxError is reported by the Python interpreter when some portion of the code is incorrect. This can include misspelled keywords, missing or too many brackets or parenthesis, incorrect operators, missing or too many quotation marks, or other conditions.

```
age = 7 + 5 = 4

File "<stdin>", line 1
SyntaxError: can't assign to operator
```

NameError

A NameError is reported by the Python interpreter when it detects a variable that is unknown. This can occur when a variable is used before it has been assigned a value or if a variable name is spelled differently than the point at which it was defined. The Python interpreter will display the line of code where the NameError was detected and indicate which name it found that was not defined.

```
misspelled_variable_name

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

ZeroDivisionError

A ZeroDivisionError is reported by the Python interpreter when it detects a division operation is being performed and the denominator (bottom number) is 0. In mathematics, dividing a number by zero has no defined value, so Python treats this as an error condition and will report a ZeroDivisionError and display the line of code where the division occurred. This can also happen if a variable is used as the denominator and its value has been set to or changed to 0.

```
numerator = 100
denominator = 0
bad_results = numerator / denominator

ZeroDivisionError: division by zero
```

Integers

An integer is a number that can be written without a fractional part (no decimal). An integer can be a positive number, a negative number or the number 0 so long as there is no decimal portion.

The number `0` represents an integer value but the same number written as `0.0` would represent a floating point number.

```
# Example integer numbers

chairs = 4
tables = 1
broken_chairs = -2
sofas = 0

# Non-integer numbers

lights = 2.5
left_overs = 0.0
```

Floating Point Numbers

Python variables can be assigned different types of data. One supported data type is the floating point number. A floating point number is a value that contains a decimal portion. It can be used to represent numbers that have fractional quantities. For example, `a = 3/5` can not be represented as an integer, so the variable `a` is assigned a floating point value of `0.6`.

```
# Floating point numbers

pi = 3.14159
meal_cost = 12.99
tip_percent = 0.20
```

Arithmetic Operations

Python supports different types of arithmetic operations that can be performed on literal numbers, variables, or some combination. The primary arithmetic operators are:

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division
- `%` for modulus (returns the remainder)
- `**` for exponentiation

```
# Arithmetic operations

result = 10 + 30
result = 40 - 10
result = 50 * 5
result = 16 / 4
result = 25 % 2
result = 5 ** 3
```

Modulo Operator %

A modulo calculation returns the remainder of a division between the first and second number. For example:

- The result of the expression `4 % 2` would result in the value 0, because 4 is evenly divisible by 2 leaving no remainder.
- The result of the expression `7 % 3` would return 1, because 7 is not evenly divisible by 3, leaving a remainder of 1.

```
# Modulo operations

zero = 8 % 4

nonzero = 12 % 5
```

String Concatenation

Python supports the joining (concatenation) of strings together using the `+` operator. The `+` operator is also used for mathematical addition operations. If the parameters passed to the `+` operator are strings, then concatenation will be performed. If the parameter passed to `+` have different types, then Python will report an error condition. Multiple variables or literal strings can be joined together using the `+` operator.

```
# String concatenation

first = "Hello "
second = "World"

result = first + second

long_result = first + second + "!"
```

Plus-Equals Operator +=

The plus-equals operator `+=` provides a convenient way to add a value to an existing variable and assign the new value back to the same variable. In the case where the variable and the value are strings, this operator performs string concatenation instead of addition.

The operation is performed in-place, meaning that any other variable which points to the variable being updated will also be updated.

```
# Plus-Equal Operator

counter = 0
counter += 10

# This is equivalent to

counter = 0
counter = counter + 10

# The operator will also perform string
concatenation

message = "Part 1 of message "
message += "Part 2 of message"
```

codecademy

Functions

Functions

Some tasks need to be performed multiple times within a program. Rather than rewrite the same code in multiple places, a function may be defined using the `def` keyword. Function definitions may include parameters, providing data input to the function. Functions may return a value using the `return` keyword followed by the value to return.

```
# Define a function my_function() with
parameter x

def my_function(x):
  return x + 1

# Invoke the function

print(my_function(2))      # Output: 3
print(my_function(3 + 5))  # Output: 9
```

Calling Functions

Python uses simple syntax to use, invoke, or *call* a preexisting function. A function can be called by writing the name of it, followed by parentheses. For example, the code provided would call the `doHomework()` method.

```
doHomework()
```

Function Indentation

Python uses indentation to identify blocks of code. Code within the same block should be indented at the same level. A Python function is one type of code block. All code under a function declaration should be indented to identify it as part of the function. There can be additional indentation within a function to handle other statements such as `for` and `if` so long as the lines are not indented less than the first line of the function code.

```
# Indentation is used to identify code
blocks

def testfunction(number):
  # This code is part of testfunction
  print("Inside the testfunction")
  sum = 0
  for x in range(number):
    # More indentation because 'for' has
a code block
    # but still part of he function
    sum += x
  return sum
print("This is not part of
testfunction")
```

Function Parameters

Sometimes functions require input to provide data for their code. This input is defined using *parameters*. *Parameters* are variables that are defined in the function definition. They are assigned the values which were passed as arguments when the function was called, elsewhere in the code.
For example, the function definition defines parameters for a character, a setting, and a skill, which are used as inputs to write the first sentence of a book.

```
def write_a_book(character, setting,
special_skill):
  print(character + " is in " +
        setting + " practicing her " +
        special_skill)
```

Multiple Parameters

Python functions can have multiple *parameters*. Just as you wouldn't go to school without both a backpack and a pencil case, functions may also need more than one input to carry out their operations.
To define a function with multiple parameters, parameter names are placed one after another, separated by commas, within the parentheses of the function definition.

```
def ready_for_school(backpack,
pencil_case):
  if (backpack == 'full' and pencil_case
== 'full'):
    print ("I'm ready for school!")
```

Function Arguments

Parameters in python are variables — placeholders for the actual values the function needs. When the function is *called*, these values are passed in as *arguments*.
For example, the arguments passed into the function `.sales()` are the "The Farmer's Market", "toothpaste", and "$1" which correspond to the parameters `grocery_store`, `item_on_sale`, and `cost`.

```
def sales(grocery_store, item_on_sale,
cost):
  print(grocery_store + " is selling " +
item_on_sale + " for " + cost)

sales("The Farmer's Market",
"toothpaste", "$1")
```

Function Keyword Arguments

Python functions can be defined with named arguments which may have default values provided. When function arguments are passed using their names, they are referred to as keyword arguments. The use of keyword arguments when calling a function allows the arguments to be passed in any order — *not* just the order that they were defined in the function. If the function is invoked without a value for a specific argument, the default value will be used.

```
def findvolume(length=1, width=1,
depth=1):
  print("Length = " + str(length))
  print("Width = " + str(width))
  print("Depth = " + str(depth))
  return length * width * depth;

findvolume(1, 2, 3)
findvolume(length=5, depth=2, width=4)
findvolume(2, depth=3, width=4)
```

Returning Value from Function

A `return` keyword is used to return a value from a Python function. The value returned from a function can be assigned to a variable which can then be used in the program.

In the example, the function `check_leap_year` returns a string which indicates if the passed parameter is a leap year or not.

```
def check_leap_year(year):
  if year % 4 == 0:
    return str(year) + " is a leap
year."
  else:
    return str(year) + " is not a leap
year."

year_to_check = 2018
returned_value =
check_leap_year(year_to_check)
print(returned_value) # 2018 is not a
leap year.
```

Returning Multiple Values

Python functions are able to return multiple values using one `return` statement. All values that should be returned are listed after the `return` keyword and are separated by commas.

In the example, the function `square_point()` returns `x_squared`, `y_squared`, and `z_squared`.

```
def square_point(x, y, z):
  x_squared = x * x
  y_squared = y * y
  z_squared = z * z
  # Return all three values:
  return x_squared, y_squared, z_squared

three_squared, four_squared,
five_squared = square_point(3, 4, 5)
```

Parameters as Local Variables

Function parameters behave identically to a function's local variables. They are initialized with the values passed into the function when it was called.

Like local variables, parameters cannot be referenced from outside the scope of the function.

In the example, the parameter `value` is defined as part of the definition of `my_function`, and therefore can only be accessed within `my_function`. Attempting to print the contents of `value` from outside the function causes an error.

```
def my_function(value):
  print(value)

# Pass the value 7 into the function
my_function(7)

# Causes an error as `value` no longer
exists
print(value)
```

Global Variables

A variable that is defined outside of a function is called a global variable. It can be accessed inside the body of a function.

In the example, the variable `a` is a global variable because it is defined outside of the function `prints_a`. It is therefore accessible to `prints_a`, which will print the value of `a`.

```
a = "Hello"

def prints_a():
  print(a)

# will print "Hello"
prints_a()
```

The Scope of Variables

In Python, a variable defined inside a function is called a local variable. It cannot be used outside of the scope of the function, and attempting to do so without defining the variable outside of the function will cause an error.

In the example, the variable `a` is defined both inside and outside of the function. When the function `f1()` is implemented, `a` is printed as `2` because it is locally defined to be so. However, when printing `a` outside of the function, `a` is printed as `5` because it is implemented outside of the scope of the function.

```
a = 5

def f1():
  a = 2
  print(a)

print(a)   # Will print 5
f1()       # Will print 2
```

codecademy

Control Flow

Equal Operator ==

The equal operator, `==`, is used to compare two values, variables or expressions to determine if they are the same.
If the values being compared are the same, the operator returns `True`, otherwise it returns `False`.
The operator takes the data type into account when making the comparison, so a string value of `"2"` is *not* considered the same as a numeric value of `2`.

```
# Equal operator

if 'Yes' == 'Yes':
  # evaluates to True
  print('They are equal')

if (2 > 1) == (5 < 10):
  # evaluates to True
  print('Both expressions give the same
result')

c = '2'
d = 2

if c == d:
  print('They are equal')
else:
  print('They are not equal')
```

Not Equals Operator !=

The Python not equals operator, `!=`, is used to compare two values, variables or expressions to determine if they are NOT the same. If they are NOT the same, the operator returns `True`. If they are the same, then it returns `False`.
The operator takes the data type into account when making the comparison so a value of `10` would NOT be equal to the string value `"10"` and the operator would return `True`. If expressions are used, then they are evaluated to a value of `True` or `False` before the comparison is made by the operator.

```
# Not Equals Operator

if "Yes" != "No":
  # evaluates to True
  print("They are NOT equal")

val1 = 10
val2 = 20

if val1 != val2:
  print("They are NOT equal")

if (10 > 1) != (10 > 1000):
  # True != False
  print("They are NOT equal")
```

Comparison Operators

In Python, *relational operators* compare two values or expressions. The most common ones are:

- `<` less than
- `>` greater than
- `<=` less than or equal to
- `>=` greater than or equal too

If the relation is sound, then the entire expression will evaluate to `True`. If not, the expression evaluates to `False`.

```
a = 2
b = 3
a < b  # evaluates to True
a > b  # evaluates to False
a >= b # evaluates to False
a <= b # evaluates to True
a <= a # evaluates to True
```

and Operator

The Python `and` operator performs a Boolean comparison between two Boolean values, variables, or expressions. If both sides of the operator evaluate to `True` then the `and` operator returns `True`. If either side (or both sides) evaluates to `False`, then the `and` operator returns `False`. A non-Boolean value (or variable that stores a value) will always evaluate to `True` when used with the `and` operator.

```
True and True      # Evaluates to True
True and False     # Evaluates to False
False and False    # Evaluates to False
1 == 1 and 1 < 2   # Evaluates to True
1 < 2 and 3 < 1    # Evaluates to False
"Yes" and 100      # Evaluates to True
```

or Operator

The Python `or` operator combines two Boolean expressions and evaluates to `True` if at least one of the expressions returns `True`. Otherwise, if both expressions are `False`, then the entire expression evaluates to `False`.

```
True or True       # Evaluates to True
True or False      # Evaluates to True
False or False     # Evaluates to False
1 < 2 or 3 < 1     # Evaluates to True
3 < 1 or 1 > 6     # Evaluates to False
1 == 1 or 1 < 2    # Evaluates to True
```

not Operator

The Python Boolean `not` operator is used in a Boolean expression in order to evaluate the expression to its inverse value. If the original expression was `True`, including the `not` operator would make the expression `False`, and vice versa.

```
not True      # Evaluates to False
not False     # Evaluates to True
1 > 2         # Evaluates to False
not 1 > 2     # Evaluates to True
1 == 1        # Evaluates to True
not 1 == 1    # Evaluates to False
```

`if` Statement

The Python `if` statement is used to determine the execution of code based on the evaluation of a Boolean expression.

- If the `if` statement expression evaluates to `True`, then the indented code following the statement is executed.
- If the expression evaluates to `False` then the indented code following the `if` statement is skipped and the program executes the next line of code which is indented at the same level as the `if` statement.

```
# if Statement

test_value = 100

if test_value > 1:
  # Expression evaluates to True
  print("This code is executed!")

if test_value > 1000:
  # Expression evaluates to False
  print("This code is NOT executed!")

print("Program continues at this
point.")
```

`else` Statement

The Python `else` statement provides alternate code to execute if the expression in an `if` statement evaluates to `False`.
The indented code for the `if` statement is executed if the expression evaluates to `True`. The indented code immediately following the `else` is executed only if the expression evaluates to `False`. To mark the end of the `else` block, the code must be unindented to the same level as the starting `if` line.

```
# else Statement

test_value = 50

if test_value < 1:
  print("Value is < 1")
else:
  print("Value is >= 1")

test_string = "VALID"

if test_string == "NOT_VALID":
  print("String equals NOT_VALID")
else:
  print("String equals something else!")
```

Boolean Values

Booleans are a data type in Python, much like integers, floats, and strings. However, booleans only have two values:

- `True`
- `False`

Specifically, these two values are of the `bool` type. Since booleans are a data type, creating a variable that holds a boolean value is the same as with other data types.

```
is_true = True
is_false = False

print(type(is_true))
# will output: <class 'bool'>
```

`elif` Statement

The Python `elif` statement allows for continued checks to be performed after an initial `if` statement. An `elif` statement differs from the `else` statement because another expression is provided to be checked, just as with the initial `if` statement. If the expression is `True`, the indented code following the `elif` is executed. If the expression evaluates to `False`, the code can continue to an optional `else` statement. Multiple `elif` statements can be used following an initial `if` to perform a series of checks. Once an `elif` expression evaluates to `True`, no further `elif` statements are executed.

```
# elif Statement

pet_type = "fish"

if pet_type == "dog":
  print("You have a dog.")
elif pet_type == "cat":
  print("You have a cat.")
elif pet_type == "fish":
  # this is performed
  print("You have a fish")
else:
  print("Not sure!")
```

Handling Exceptions in Python

A `try` and `except` block can be used to handle error in code block. Code which may raise an error can be written in the `try` block. During execution, if that code block raises an error, the rest of the `try` block will cease executing and the `except` code block will execute.

```
def check_leap_year(year):
  is_leap_year = False
  if year % 4 == 0:
    is_leap_year = True

try:
  check_leap_year(2018)
  print(is_leap_year)
  # The variable is_leap_year is
declared inside the function
except:
  print('Your code raised an error!')
```

codecademy

Lists

Lists

In Python, lists are ordered collections of items that allow for easy use of a set of data.

List values are placed in between square brackets `[ ]`, separated by commas. It is good practice to put a space between the comma and the next value. The values in a list do not need to be unique (the same value can be repeated).

Empty lists do not contain any values within the square brackets.

```
primes = [2, 3, 5, 7, 11]
print(primes)

empty_list = []
```

Python Lists: Data Types

In Python, lists are a versatile data type that can contain multiple different data types within the same square brackets. The possible data types within a list include numbers, strings, other objects, and even other lists.

```
numbers = [1, 2, 3, 4, 10]
names = ['Jenny', 'Sam', 'Alexis']
mixed = ['Jenny', 1, 2]
list_of_lists = [['a', 1], ['b', 2]]
```

Aggregating Iterables Using `zip()`

In Python, data types that can be iterated (called iterables) can be used with the `zip()` function to aggregate data. The `zip()` function takes iterables, aggregates corresponding elements based on the iterables passed in, and returns an iterator. Each element of the returned iterator is a tuple of values.

As shown in the example, `zip()` is aggregating the data between the owners' names and the dogs' names to match the owner to their dogs. `zip()` returns an iterator containing the data based on what the user passes to the function. Empty iterables passed in will result in an empty iterator. To view the contents of the iterator returned from `zip()`, we can cast it as a list by using the `list()` function and printing the results.

```
owners_names = ['Jenny', 'Sam',
'Alexis']
dogs_names = ['Elphonse', 'Dr. Doggy
DDS', 'Carter']
owners_dogs = zip(owners_names,
dogs_names)
print(list(owners_dogs))
# Result: [('Jenny', 'Elphonse'),
('Sam', 'Dr.Doggy DDS'), ('Alexis',
'Carter')]
```

List Method `.append()`

In Python, you can add values to the end of a list using the `.append()` method. This will place the object passed in as a new element at the very end of the list. Printing the list afterwards will visually show the appended value. This `.append()` method is *not* to be confused with returning an entirely new list with the passed object.

```
orders = ['daisies', 'periwinkle']
orders.append('tulips')
print(orders)
# Result: ['daisies', 'periwinkle',
'tulips']
```

Adding Lists Together

In Python, lists can be added to each other using the plus symbol `+`. As shown in the code block, this will result in a new list containing the same items in the same order with the first list's items coming first.
Note: This will not work for adding one item at a time (use `.append()` method). In order to add one item, create a new list with a single value and then use the plus symbol to add the list.

```
items = ['cake', 'cookie', 'bread']
total_items = items + ['biscuit',
'tart']
print(total_items)
# Result: ['cake', 'cookie', 'bread',
'biscuit', 'tart']
```

Determining List Length with `len()`

The Python `len()` function can be used to determine the number of items found in the list it accepts as an argument.

```
knapsack = [2, 4, 3, 7, 10]
size = len(knapsack)
print(size)
# Output: 5
```

Zero-Indexing

In Python, list index begins at zero and ends at the length of the list minus one. For example, in this list, `'Andy'` is found at index `2`.

```
names = ['Roger', 'Rafael', 'Andy',
'Novak']
```

List Indices

Python list elements are ordered by *index*, a number referring to their placement in the list. List indices start at 0 and increment by one.
To access a list element by index, square bracket notation is used: `list[index]`.

```
berries = ["blueberry", "cranberry",
"raspberry"]

berries[0]   # "blueberry"
berries[2]   # "raspberry"
```

List Slicing

A *slice*, or sub-list of Python list elements can be selected from a list using a colon-separated starting and ending point.
The syntax pattern is `myList[START_NUMBER:END_NUMBER]`. The slice will include the `START_NUMBER` index, and everything until but excluding the `END_NUMBER` item.
When slicing a list, a new list is returned, so if the slice is saved and then altered, the original list remains the same.

```
tools = ['pen', 'hammer', 'lever']
tools_slice = tools[1:3] # ['hammer',
'lever']
tools_slice[0] = 'nail'

# Original list is unaltered:
print(tools) # ['pen', 'hammer',
'lever']
```

List Item Ranges Including First or Last Item

In Python, when selecting a range of list items, if the first item to be selected is at index `0`, no index needs to be specified before the `:`. Similarly, if the last item selected is the last item in the list, no index needs to be specified after the `:`.

```
items = [1, 2, 3, 4, 5, 6]

# All items from index `0` to `3`
print(items[:4])

# All items from index `2` to the last item, inclusive
print(items[2:])
```

Negative List Indices

Negative indices for lists in Python can be used to reference elements in relation to the end of a list. This can be used to access single list elements or as part of defining a list range. For instance:

- To select the last element, `my_list[-1]`.
- To select the last three elements, `my_list[-3:]`.
- To select everything except the last two elements, `my_list[:-2]`.

```
soups = ['minestrone', 'lentil', 'pho', 'laksa']
soups[-1]   # 'laksa'
soups[-3:]  # 'lentil', 'pho', 'laksa'
soups[:-2]  # 'minestrone', 'lentil'
```

List Method `.count()`

The `.count()` Python list method searches a list for whatever search term it receives as an argument, then returns the number of matching entries found.

```
backpack = ['pencil', 'pen', 'notebook', 'textbook', 'pen', 'highlighter', 'pen']
numPen = backpack.count('pen')
print(numPen)
# Output: 3
```

List Method `.sort()`

The `.sort()` Python list method will sort the contents of whatever list it is called on. Numerical lists will be sorted in ascending order, and lists of Strings will be sorted into alphabetical order. It modifies the original list, and has no return value.

```
exampleList = [4, 2, 1, 3]
exampleList.sort()
print(exampleList)
# Output: [1, 2, 3, 4]
```

`sorted()` Function

The Python `sorted()` function accepts a list as an argument, and will return a new, sorted list containing the same elements as the original. Numerical lists will be sorted in ascending order, and lists of Strings will be sorted into alphabetical order. It does not modify the original, unsorted list.

```
unsortedList = [4, 2, 1, 3]
sortedList = sorted(unsortedList)
print(sortedList)
# Output: [1, 2, 3, 4]
```

codecademy

Loops

Python For Loop

A Python `for` loop can be used to iterate over a list of items and perform a set of actions on each item.

The syntax of a `for` loop consists of assigning a temporary value to a variable on each successive iteration.

When writing a `for` loop, remember to properly indent each action, otherwise an `IndentationError` will result.

```
for <temporary variable> in <list variable>:
  <action statement>
  <action statement>

#each num in nums will be printed below
nums = [1,2,3,4,5]
for num in nums:
  print(num)
```

Python for Loops

Python `for` loops can be used to iterate over and perform an action one time for each element in a list.

Proper `for` loop syntax assigns a temporary value, the current item of the list, to a variable on each successive iteration: `for <temporary value> in <a list>:`

`for` loop bodies must be indented to avoid an `IndentationError`.

```
dog_breeds = ["boxer", "bulldog", "shiba inu"]

# Print each breed:
for breed in dog_breeds:
  print(breed)
```

Python Loops with range().

In Python, a `for` loop can be used to perform an action a specific number of times in a row.

The `range()` function can be used to create a list that can be used to specify the number of iterations in a `for` loop.

```
# Print the numbers 0, 1, 2:
for i in range(3):
  print(i)

# Print "WARNING" 3 times:
for i in range(3):
  print("WARNING")
```

Infinite Loop

An infinite loop is a loop that never terminates. Infinite loops result when the conditions of the loop prevent it from terminating. This could be due to a typo in the conditional statement within the loop or incorrect logic. To interrupt a Python program that is running forever, press the `Ctrl` and `C` keys together on your keyboard.

break Keyword

In a loop, the `break` keyword escapes the loop, regardless of the iteration number. Once `break` executes, the program will continue to execute after the loop.
In this example, the output would be:

- `0`
- `254`
- `2`
- `Negative number detected!`

```
numbers = [0, 254, 2, -1, 3]

for num in numbers:
  if (num < 0):
    print("Negative number detected!")
    break
  print(num)

# 0
# 254
# 2
# Negative number detected!
```

The Python continue Keyword

In Python, the `continue` keyword is used inside a loop to skip the remaining code inside the loop code block and begin the next loop iteration.

```
big_number_list = [1, 2, -1, 4, -5, 5,
2, -9]

# Print only positive numbers:
for i in big_number_list:
  if i < 0:
    continue
  print(i)
```

Python while Loops

In Python, a `while` loop will repeatedly execute a code block as long as a condition evaluates to `True`. The condition of a `while` loop is always checked first before the block of code runs. If the condition is not met initially, then the code block will never run.

```
# This loop will only run 1 time
hungry = True
while hungry:
  print("Time to eat!")
  hungry = False

# This loop will run 5 times
i = 1
while i < 6:
  print(i)
  i = i + 1
```

Python Nested Loops

In Python, loops can be *nested* inside other loops. Nested loops can be used to access items of lists which are inside other lists. The item selected from the outer loop can be used as the list for the inner loop to iterate over.

```
groups = [["Jobs", "Gates"], ["Newton", "Euclid"], ["Einstein", "Feynman"]]

# This outer loop will iterate over each list in the groups list
for group in groups:
  # This inner loop will go through each name in each list
  for name in group:
    print(name)
```

Python List Comprehension

Python list comprehensions provide a concise way for creating lists. It consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses: `[EXPRESSION for ITEM in LIST <if CONDITIONAL>]`.
The expressions can be anything - any kind of object can go into a list.
A list comprehension always returns a list.

```
# List comprehension for the squares of all even numbers between 0 and 9
result = [x**2 for x in range(10) if x % 2 == 0]

print(result)
# [0, 4, 16, 36, 64]
```

codecademy

Strings

Strings

In computer science, sequences of characters are referred to as *strings*. Strings can be any length and can include any character such as letters, numbers, symbols, and whitespace (spaces, tabs, new lines).

Indexing and Slicing Strings

Python strings can be indexed using the same notation as lists, since strings are lists of characters. A single character can be accessed with bracket notation (`[index]`), or a substring can be accessed using slicing (`[start:end]`).
Indexing with negative numbers counts from the end of the string.

```
str = 'yellow'
str[1]     # => 'e'
str[-1]    # => 'w'
str[4:6]   # => 'ow'
str[:4]    # => 'yell'
str[-3:]   # => 'low'
```

String Concatenation

To combine the content of two strings into a single string, Python provides the `+` operator. This process of joining strings is called concatenation.

```
x = 'One fish, '
y = 'two fish.'

z = x + y

print(z)
# Output: One fish, two fish.
```

Built-in Function `len()`

In Python, the built-in `len()` function can be used to determine the length of an object. It can be used to compute the length of strings, lists, sets, and other countable objects.

```
length = len("Hello")
print(length)
# Output: 5

colors = ['red', 'yellow', 'green']
print(len(colors))
# Output: 3
```

IndexError

When indexing into a string in Python, if you try to access an index that doesn't exist, an `IndexError` is generated. For example, the following code would create an `IndexError` :

```
fruit = "Berry"
indx = fruit[6]
```

Immutable strings

Strings are immutable in Python. This means that once a string has been defined, it can't be changed.
There are no mutating methods for strings. This is unlike data types like lists, which can be modified once they are created.

Escaping Characters

Backslashes (`\`) are used to escape characters in a Python string.
For instance, to print a string with quotation marks, the given code snippet can be used.

```
txt = "She said \"Never let go\"."
print(txt) # She said "Never let go".
```

Iterate String

To iterate through a string in Python, "for...in" notation is used.

```
str = "hello"
for c in str:
  print(c)

# h
# e
# l
# l
# o
```

The `in` Syntax

The `in` syntax is used to determine if a letter or a substring exists in a string. It returns `True` if a match is found, otherwise `False` is returned.

```
game = "Popular Nintendo Game: Mario Kart"

print("l" in game) # Prints: True
print("x" in game) # Prints: False
```

String Method `.lower()`

The string method `.lower()` returns a string with all uppercase characters converted into lowercase.

```
greeting = "Welcome To Chili's"

print(greeting.lower())
# Prints: welcome to chili's
```

String Method `.upper()`

The string method `.upper()` returns the string with all lowercase characters converted to uppercase.

```
dinosaur = "T-Rex"

print(dinosaur.upper())
# Prints: T-REX
```

String Method `.title()`

The string method `.title()` returns the string in title case. With title case, the first character of each word is capitalized while the rest of the characters are lowercase.

```
my_var = "dark knight"
print(my_var.title())

# Prints: Dark Knight
```

String Method `.split()`

The string method `.split()` splits a string into a list of items:

- If no argument is passed, the default behavior is to split on whitespace.
- If an argument is passed to the method, that value is used as the delimiter on which to split the string.

```
text = "Silicon Valley"

print(text.split())
# Prints: ['Silicon', 'Valley']

print(text.split('i'))
# Prints: ['S', 'l', 'con Valley']
```

String Method `.join()`

The string method `.join()` concatenates a list of strings together to create a new string joined with the desired delimiter.
The `.join()` method is run on the delimiter and the array of strings to be concatenated together is passed in as an argument.

```
x = "-".join(["Codecademy", "is",
"awesome"])

print(x)
# Prints: Codecademy-is-awesome
```

String Method `.strip()`

The string method `.strip()` can be used to remove characters from the beginning and end of a string.
A string argument can be passed to the method, specifying the set of characters to be stripped. With no arguments to the method, whitespace is removed.

```
text1 = '   apples and oranges   '
text1.strip()       # => 'apples and
oranges'

text2 = '...+...lemons and limes...-...'

# Here we strip just the "." characters
text2.strip('.')    # => '+...lemons and
limes...-'

# Here we strip both "." and "+"
characters
text2.strip('.+')   # => 'lemons and
limes...-'

# Here we strip ".", "+", and "-"
characters
text2.strip('.+-')  # => 'lemons and
limes'
```

String replace

The `.replace()` method is used to replace the occurence of the first argument with the second argument within the string.
The first argument is the old substring to be replaced, and the second argument is the new substring that will replace every occurence of the first one within the string.

```
fruit = "Strawberry"
print(fruit.replace('r', 'R'))

# StRawbeRRy
```

Python string method `.find()`

The Python string method `.find()` returns the index of the first occurrence of the string passed as the argument. It returns `-1` if no occurrence is found.

```
mountain_name = "Mount Kilimanjaro"
print(mountain_name.find("o")) # Prints
1 in the console.
```

Python String .format()

The Python string method `.format()` replaces empty brace (`{}`) placeholders in the string with its arguments.
If keywords are specified within the placeholders, they are replaced with the corresponding named arguments to the method.

```
msg1 = 'Fred scored {} out of {}
points.'
msg1.format(3, 10)
# => 'Fred scored 3 out of 10 points.'

msg2 = 'Fred {verb} a {adjective}
{noun}.'
msg2.format(adjective='fluffy',
verb='tickled', noun='hamster')
# => 'Fred tickled a fluffy hamster.'
```

codecademy

Modules

Import Python Modules

The Python **import** statement can be used to import Python modules from other files.
Modules can be imported in three different ways: `import module`, `from module import functions`, or `from module import *`. `from module import *` is discouraged, as it can lead to a cluttered local namespace and can make the namespace unclear.

```
# Three different ways to import
modules:
# First way
import module
module.function()

# Second way
from module import function
function()

# Third way
from module import *
function()
```

Module importing

In Python, you can import and use the content of another file using `import filename`, provided that it is in the same folder as the current file you are writing.

```
# file1 content
# def f1_function():
#         return "Hello World"

# file2
import file1

# Now we can use f1_function, because we
imported file1
f1_function()
```

Aliasing with ‘as’ keyword

In Python, the `as` keyword can be used to give an alternative name as an alias for a Python module or function.

```
# Aliasing matplotlib.pyplot as plt
from matplotlib import pyplot as plt
plt.plot(x, y)

# Aliasing calendar as c
import calendar as c
print(c.month_name[1])
```

Date and Time in Python

Python provides a module named `datetime` to deal with dates and times.

It allows you to set `date` , `time` or both `date` and `time` using the `date()` , `time()` and `datetime()` functions respectively, after importing the `datetime` module .

```
import datetime
feb_16_2019 = datetime.date(year=2019, month=2, day=16)
feb_16_2019 = datetime.date(2019, 2, 16)
print(feb_16_2019) #2019-02-16

time_13_48min_5sec = datetime.time(hour=13, minute=48, second=5)
time_13_48min_5sec = datetime.time(13, 48, 5)
print(time_13_48min_5sec) #13:48:05

timestamp= datetime.datetime(year=2019, month=2, day=16, hour=13, minute=48, second=5)
timestamp = datetime.datetime(2019, 2, 16, 13, 48, 5)
print (timestamp) #2019-01-02 13:48:05
```

`random.randint()` and `random.choice()`

In Python, the `random` module offers methods to simulate non-deterministic behavior in selecting a random number from a range and choosing a random item from a list.

The `randint()` method provides a uniform random selection from a range of integers. The `choice()` method provides a uniform selection of a random element from a sequence.

```
# Returns a random integer N in a given range, such that start <= N <= end
# random.randint(start, end)
r1 = random.randint(0, 10)
print(r1) # Random integer where 0 <= r1 <= 10

# Prints a random element from a sequence
seq = ["a", "b", "c", "d", "e"]
r2 = random.choice(seq)
print(r2) # Random element in the sequence
```

codecademy

Dictionaries

Python dictionaries

A python dictionary is an unordered collection of items. It contains data as a set of key: value pairs.

```
my_dictionary = {1: "L.A. Lakers", 2:
"Houston Rockets"}
```

Syntax of the Python dictionary

The syntax for a Python dictionary begins with the left curly brace (`{`), ends with the right curly brace (`}`), and contains zero or more `key : value` items separated by commas (`,`). The `key` is separated from the `value` by a colon (`:`).

```
roaster = {"q1": "Ashley", "q2":
"Dolly"}
```

Dictionary value types

Python allows the *values* in a dictionary to be any type – string, integer, a list, another dictionary, boolean, etc. However, *keys* must always be an immutable data type, such as strings, numbers, or tuples.
In the example code block, you can see that the keys are strings or numbers (int or float). The values, on the other hand, are many varied data types.

```
dictionary = {
  1: 'hello',
  'two': True,
  '3': [1, 2, 3],
  'Four': {'fun': 'addition'},
  5.0: 5.5
}
```

Accessing and writing data in a Python dictionary

Values in a Python dictionary can be accessed by placing the key within square brackets next to the dictionary. Values can be written by placing key within square brackets next to the dictionary and using the assignment operator (`=`). If the key already exists, the old value will be overwritten. Attempting to access a value with a key that does not exist will cause a `KeyError` .
To illustrate this review card, the second line of the example code block shows the way to access the value using the key `"song"` . The third line of the code block overwrites the value that corresponds to the key `"song"` .

```
my_dictionary = {"song": "Estranged",
"artist": "Guns N' Roses"}
print(my_dictionary["song"])
my_dictionary["song"] = "Paradise City"
```

Merging Dictionaries with the `.update()` Method in Python

Given two dictionaries that need to be combined, Python makes this easy with the `.update()` function.

For `dict1.update(dict2)`, the key-value pairs of `dict2` will be written into the `dict1` dictionary.

For keys in *both* `dict1` and `dict2`, the value in `dict1` will be overwritten by the corresponding value in `dict2`.

```
dict1 = {'color': 'blue', 'shape':
'circle'}
dict2 = {'color': 'red', 'number': 42}

dict1.update(dict2)

# dict1 is now {'color': 'red', 'shape':
'circle', 'number': 42}
```

get() Method for Dictionary

Python provides a `.get()` method to access a `dictionary` value if it exists. This method takes the `key` as the first argument and an optional default value as the second argument, and it returns the value for the specified `key` if `key` is in the dictionary. If the second argument is not specified and `key` is not found then `None` is returned.

```
# without default
{"name": "Victor"}.get("name")
# returns "Victor"

{"name": "Victor"}.get("nickname")
# returns None

# with default
{"name": "Victor"}.get("nickname",
"nickname is not a key")
# returns "nickname is not a key"
```

The `.pop()` Method for Dictionaries in Python

Python dictionaries can remove key-value pairs with the `.pop()` method. The method takes a key as an argument and removes it from the dictionary. At the same time, it also returns the value that it removes from the dictionary.

```
famous_museums = {'Washington':
'Smithsonian Institution', 'Paris': 'Le
Louvre', 'Athens': 'The Acropolis
Museum'}
famous_museums.pop('Athens')
print(famous_museums) # {'Washington':
'Smithsonian Institution', 'Paris': 'Le
Louvre'}
```

Dictionary accession methods

When trying to look at the information in a Python dictionary, there are multiple methods that access the dictionary and return lists of its contents.

`.keys()` returns the keys (the first object in the key-value pair), `.values()` returns the values (the second object in the key-value pair), and `.items()` returns both the keys and the values as a tuple.

```
ex_dict = {"a": "anteater", "b":
"bumblebee", "c": "cheetah"}

ex_dict.keys()
# ["a","b","c"]

ex_dict.values()
# ["anteater", "bumblebee", "cheetah"]

ex_dict.items()
# [("a","anteater"),("b","bumblebee"),
("c","cheetah")]
```

codecademy

Files

Python File Object

A Python file object is created when a file is opened with the `open()` function. You can associate this file object with a variable when you open a file using the `with` and `as` keywords. For example:

```
with open('somefile.txt') as
file_object:
```

You can then print the content of the file object, `file_object` with `print()`.

```
print(file_object)
```

You might see something like this on the output terminal:

```
<_io.TextIOWrapper
name='somefile.txt' mode='r'
encoding='UTF-8'>
```

Python Read Method

After a file is opened with `open()` returning a file object, call the `.read()` method of the file object to return the entire file content as a Python string. Executing the following Python code:

```
with open('mystery.txt') as
text_file:
  text_data = text_file.read()
print(text_data)
```

will produce a string containing the entire content of the read file:

```
Mystery solved.
Congratulations!
```

Python Readlines Method

Instead of reading the entire content of a file, you can read a single line at a time. Instead of `.read()` which returns a string, call `.readlines()` to return a list of strings, each representing an individual line in the file. Calling this code:

```
with open('lines.txt') as
file_object:
  file_data =
file_object.readlines()
print(file_data)
```

returns a list of strings in `file_data`:

```
['1. Learn Python.\n', '2. Work
hard.\n', '3. Graduate.']
```

Iterating over the list, `file_data`, and printing it:

```
for line in file_data:
  print(line)
```

outputs:

```
1. Learn Python.

2. Work hard.

3. Graduate.
```

Python Readline Method

To read only one line instead of multiple lines in a Python file, use the method `.readline()` on a file object that is returned from the `open()` function. Every subsequent `.readline()` will extract the next line in the file if it exists.

```
with open('story.txt') as
story_object:
  print(story_object.readline())
```

will print only the first line in `story.txt`.

Python Write To File

By default, a file when opened with `open()` is only for reading. A second argument `'r'` is passed to it by default. To write to a file, first open the file with write permission via the `'w'` argument. Then use the `.write()` method to write to the file. If the file already exists, all prior content will be overwritten.

```
with open('diary.txt','w') as
diary:
  diary.write('Special events for
today')
```

Python Append To File

Writing to an opened file with the `'w'` flag overwrites all previous content in the file. To avoid this, we can append to a file instead. Use the `'a'` flag as the second argument to `open()`. If a file doesn't exist, it will be created for append mode.

```
with open('shopping.txt', 'a') as
shop:
  shop.write('Tomatoes,
cucumbers, celery\n')
```

Class csv.DictWriter

In Python, the `csv` module implements classes to read and write tabular data in *CSV* format. It has a class `DictWriter` which operates like a regular writer but maps a dictionary onto output rows. The keys of the dictionary are column names while values are actual data.

The `csv.DictWriter` constructor takes two arguments. The first is the open file handler that the CSV is being written to. The second named parameter, `fieldnames`, is a list of field names that the CSV is going to handle.

```
# An example of csv.DictWriter
import csv

with open('companies.csv', 'w') as
csvfile:
  fieldnames = ['name', 'type']
  writer = csv.DictWriter(csvfile,
fieldnames=fieldnames)
  writer.writeheader()
  writer.writerow({'name': 'Codecademy',
'type': 'Learning'})
  writer.writerow({'name': 'Google',
'type': 'Search'})

"""
After running the above code,
companies.csv will contain the following
information:

name,type
Codecademy,Learning
Google,Search
"""
```

codecademy

Classes

Python type() function

The Python `type()` function returns the data type of the argument passed to it.

```
a = 1
print type(a) # <type 'int'>

a = 1.1
print type(a) # <type 'float'>

a = 'b'
print type(a) # <type 'str'>

a = None
print type(a) # <type 'NoneType'>
```

Python class

In Python, a class is a template for a data type. A class can be defined using the `class` keyword.

```
# Defining a class
class Animal:
  def __init__(self, name,
number_of_legs):
    self.name = name
    self.number_of_legs = number_of_legs
```

Instantiate Python Class

In Python, a class needs to be instantiated before use. As an analogy, a class can be thought of as a blueprint (Car), and an instance is an actual implementation of the blueprint (Ferrari).

```
class Car:
  "This is an empty class"
  pass

# Class Instantiation
ferrari = Car()
```

`__main__` in Python

In Python, `__main__` is an identifier used to reference the current file context. When a module is read from standard input, a script, or from an interactive prompt, its `__name__` is set equal to `__main__`.
Suppose we create an instance of a class called `CoolClass`. Printing the `type()` of the instance will result in:

```
<class '__main__.CoolClass'>
```

This means that the class `CoolClass` was defined in the current script file.

Python Class Variables

In Python, class variables are defined outside of all methods and have the same value for every instance of the class.
Class variables are accessed with the `instance.variable` or `class_name.variable` syntaxes.

```
class my_class:
  class_variable = "I am a Class
Variable!"

x = my_class()
y = my_class()

print(x.class_variable) #I am a Class
Variable!
print(y.class_variable) #I am a Class
Variable!
```

Python class methods

In Python, *methods* are functions that are defined as part of a class. It is common practice that the first argument of any method that is part of a class is the actual object calling the method. This argument is usually called **self**.

```
# Dog class
class Dog:
  # Method of the class
  def bark(self):
    print("Ham-Ham")

# Create a new instance
charlie = Dog()

# Call the method
charlie.bark()
# This will output "Ham-Ham"
```

Python dir() function

In Python, the built-in `dir()` function, without any argument, returns a list of all the attributes in the current scope.

With an object as argument, `dir()` tries to return all valid object attributes.

```
class Employee:
  def __init__(self, name):
    self.name = name

  def print_name(self):
    print("Hi, I'm " + self.name)


print(dir())
# ['Employee', '__builtins__',
'__doc__', '__file__', '__name__',
'__package__', 'new_employee']

print(dir(Employee))
# ['__doc__', '__init__', '__module__',
'print_name']
```

Python repr method

The Python `__repr__()` method is used to tell Python what the *string representation* of the class should be. It can only have one parameter, `self`, and it should return a string.

```
class Employee:
  def __init__(self, name):
    self.name = name

  def __repr__(self):
    return self.name

john = Employee('John')
print(john) # John
```

Python init method

In Python, the `.__init__()` method is used to initialize a newly created object. It is called every time the class is instantiated.

```
class Animal:
  def __init__(self, voice):
    self.voice = voice

# When a class instance is created, the
instance variable
# 'voice' is created and set to the
input value.
cat = Animal('Meow')
print(cat.voice) # Output: Meow

dog = Animal('Woof')
print(dog.voice) # Output: Woof
```

Python Inheritance

Subclassing in Python, also known as “inheritance”, allows classes to share the same attributes and methods from a parent or superclass. Inheritance in Python can be accomplished by putting the superclass name between parentheses after the subclass or child class name.

In the example code block, the `Dog` class subclasses the `Animal` class, inheriting all of its attributes.

```
class Animal:
  def __init__(self, name, legs):
    self.name = name
    self.legs = legs

class Dog(Animal):
  def sound(self):
    print("Woof!")

Yoki = Dog("Yoki", 4)
print(Yoki.name) # YOKI
print(Yoki.legs) # 4
Yoki.sound() # Woof!
```

User-defined exceptions in Python

In Python, new exceptions can be defined by creating a new class which has to be derived, either directly or indirectly, from Python’s **Exception** class.

```
class CustomError(Exception):
  pass
```

Python issubclass() Function

The Python `issubclass()` built-in function checks if the first argument is a subclass of the second argument.

In the example code block, we check that `Member` is a subclass of the `Family` class.

```
class Family:
  def type(self):
    print("Parent class")

class Member(Family):
  def type(self):
    print("Child class")

print(issubclass(Member, Family)) # True
```

Method Overriding in Python

In Python, inheritance allows for method overriding, which lets a child class change and redefine the implementation of methods already defined in its parent class.

The following example code block creates a `ParentClass` and a `ChildClass` which both define a `print_test()` method.

As the `ChildClass` inherits from the `ParentClass`, the method `print_test()` will be overridden by `ChildClass` such that it prints the word “Child” instead of “Parent”.

```
class ParentClass:
  def print_self(self):
    print("Parent")

class ChildClass(ParentClass):
  def print_self(self):
    print("Child")

child_instance = ChildClass()
child_instance.print_self() # Child
```

Super() Function in Python Inheritance

Python's `super()` function allows a subclass to invoke its parent's version of an overridden method.

```
class ParentClass:
  def print_test(self):
    print("Parent Method")

class ChildClass(ParentClass):
  def print_test(self):
    print("Child Method")
    # Calls the parent's version of
print_test()
    super().print_test()

child_instance = ChildClass()
child_instance.print_test()
# Output:
# Child Method
# Parent Method
```

Polymorphism in Python

When two Python classes offer the same set of methods with different implementations, the classes are *polymorphic* and are said to have the same *interface*. An interface in this sense might involve a common inherited class and a set of overridden methods. This allows using the two objects in the same way regardless of their individual types.
When a child class overrides a method of a parent class, then the type of the object determines the version of the method to be called. If the object is an instance of the child class, then the child class version of the overridden method will be called. On the other hand, if the object is an instance of the parent class, then the parent class version of the method is called.

```
class ParentClass:
  def print_self(self):
    print('A')

class ChildClass(ParentClass):
  def print_self(self):
    print('B')

obj_A = ParentClass()
obj_B = ChildClass()

obj_A.print_self() # A
obj_B.print_self() # B
```

+ Operator

In Python, the `+` operation can be defined for a user-defined class by giving that class an `.__add()__` method.

```
class A:
  def __init__(self, a):
    self.a = a
  def __add__(self, other):
    return self.a + other.a

obj1 = A(5)
obj2 = A(10)
print(obj1 + obj2) # 15
```

Dunder methods in Python

Dunder methods, which stands for “Double Under” (Underscore) methods, are special methods which have double underscores at the beginning and end of their names.

We use them to create functionality that can’t be represented as a normal method, and resemble native Python data type interactions. A few examples for dunder methods are: `__init__` , `__add__` , `__len__` , and `__iter__` .

The example code block shows a class with a definition for the `__init__` dunder method.

```
class String:
  # Dunder method to initialize object
  def __init__(self, string):
    self.string = string

string1 = String("Hello World!")
print(string1.string) # Hello World!
```

codecademy

Function Arguments

Python function default return value

If we do not not specify a return value for a Python function, it returns `None`. This is the default behaviour.

```
# Function returning None
def my_function(): pass

print(my_function())

#Output
None
```

Python variable None check

To check if a Python variable is None we can make use of the statement `variable is None`.
If the above statement evaluates to True, the `variable` value is None.

```
# Variable check for None
if variable_name is None:
    print "variable is None"
else:
    print "variable is NOT None"
```

Default argument is fallback value

In Python, a default parameter is defined with a fallback value as a default argument. Such parameters are optional during a function call. If no argument is provided, the default value is used, and if an argument is provided, it will overwrite the default value.

```
def greet(name, msg="How do you do?"):
  print("Hello ", name + ', ' + msg)

greet("Ankit")
greet("Ankit", "How do you do?")

"""
this code will print the following for
both the calls -
`Hello  Ankit, How do you do?`
"""
```

Python Default Arguments

A Python function cannot define a default argument in its signature before any required parameters that do not have a default argument. Default arguments are ones set using the form `parameter=value`. If no input value is provided for such arguments, it will take on the default value.

```
# Correct order of declaring default argments in a function
def greet(name, msg = "Good morning!"):
  print("Hello ", name + ', ' + msg)

# The function can be called in the following ways
greet("Ankit")
greet("Kyla","How are you?")

# The following function definition would be incorrect
def greet(msg = "Good morning!", name):
  print("Hello ", name + ', ' + msg)
# It would cause a "SyntaxError: non-default argument follows default argument"
```

Python function arguments

A function can be called using the argument name as a keyword instead of relying on its positional value. Functions define the argument names in its composition then those names can be used when calling the function.

```
# The function will take arg1 and arg2
def func_with_args(arg1, arg2):
  print(arg1 + ' ' + arg2)

func_with_args('First', 'Second')
# Prints:
# First Second

func_with_args(arg2='Second', arg1='First')
# Prints
# First Second
```

Mutable Default Arguments

Python's default arguments are evaluated only once when the function is defined, not each time the function is called. This means that if a mutable default argument is used and is mutated, it is mutated for all future calls to the function as well. This leads to buggy behaviour as the programmer expects the default value of that argument in each function call.

```
# Here, an empty list is used as a
default argument of the function.
def append(number, number_list=[]):
  number_list.append(number)
  print(number_list)
  return number_list

# Below are 3 calls to the `append`
function and their expected and actual
outputs:
append(5) # expecting: [5], actual: [5]
append(7) # expecting: [7], actual: [5,
7]
append(2) # expecting: [2], actual: [5,
7, 2]
```