Code First Girls Nanodegree—Foundation Module Theory Assignment

Robyn Seymour-Jones

Contents

1 | Python theory questions (30 points) 1

2 | Python string methods (29 points) 16

3 | Python list methods (11 points) 20

4 | Python tuple methods (2 points) 22

5 | Python dictionary methods (11 points) 23

6 | Python set methods (12 points) 26

7 | Python file methods (11 points) 28

1 | Python theory questions (30 points)

1.1 What is Python and what are its main features?

Computers work in binary (0s and 1s) but this is difficult for a human to understand. Programming languages have been developed to make it easier for humans to write instructions to, and read replies from, a computer. They consist of human-readable syntax that is translated to the computer's binary language by a compiler or interpreter.

There are many programming languages that make communicating with a computer easier for humans. Each one's popularity varies through time as requirements change and new languages are developed. **Python is currently one of the most popular programming languages** and is used for a variety of purposes including programming software applications, web development, data analysis, data visualisation and machine learning.

Main Python features:

- **Interpreted/scripted**—Python uses a program (the Python interpreter, which itself is written in the programming language C) to translate commands and interpret them directly from the source code without a compilation step. You don't need to keep track of memory management.
- **Dynamically-typed**—The word "type" refers to the noun (e.g. data type) rather than the verb "to type". As there is no compilation step, the Python interpreter does type checking during runtime (i.e. when the code is run). You do not have to explicitly declare the type of an object.
- **Strongly-typed**—Python does little implicit type conversion. For example, it will convert an int to a float without you explicitly telling it to do so but it will not convert an int to a string unless you tell it to. (Languages that will do this are known as untyped or weakly/loosely typed, e.g. JavaScript).
- **High-level**—A high-level programming language is closer to human language and further from machine language than a low-level programming

language, i.e. they have a higher level of abstraction from machine language. This means there are fewer things the programmer has to worry about. For example, memory management is automated. This makes it easier to learn.

- **Open source**—The source code is **free** and **openly available** for anyone to download, modify and distribute.
- **Terse**—As Python code is terse, more can be done with fewer lines of code.
- **Readable**—Python syntax is clear and readable as it is designed to be close to actual English.
- **Runs almost everywhere**—Python is platform independent because it is an interpreted language using a virtual machine rather than the CPU.
- **Enjoyable to use**—People enjoy it so they want to use it! In turn, this makes it popular and leads to a large community of users who develop lots of libraries as well as learning resources.

1.2 Discuss the difference between Python 2 and Python 3

Python 2 is an older version of Python 3 that is no longer being updated. As of January 1st 2020, no bug fixes or changes are being made (even if there are catastrophic security problems); Python 2 is no longer supported. The last version of Python 2 is version 2.7. Many legacy applications have been ported to Python 3, however, there are still many that run on Python 2, so it is useful to know the differences in case you encounter them. They look similar but there are some key differences:

Difference	Python 2	Python 3
Type system	Untyped	Strongly typed
Libraries	Many libraries developed for Python 2 are not compatible with Python 3.	Many new Python 3 libraries are not backwards compatible with Python 2.
Print	Can print without using parentheses, e.g. `print 3`	Requires parentheses to print, e.g. `print(3)`
Strings	Strings stored as ASCII by default.	Strings stored as Unicode by default.

String formatting	Only accepts old style string formatting, e.g.
	`name = 'Robyn'`
	`print 'Hello %' % name`
		Accepts old and new styles, e.g.
		`name = Robyn`
		`print('Hello %' % name)`
		`print('Hello {}'.format(name))`
		`print(f'Hello {name}')`

Difference	Python 2	Python 3

Division	Rounds all divisions down (floor division), e.g.
	`>>> 5 / 2`
	`2`
		Returns expected result, e.g.
		`>>> 5 / 2`
		`2.5`

1.3 What is PEP 8?

Python Enhancement Proposal (PEP) 8 is the most recent **style guide for writing Python code**. It provides guidance on the best practices to use when writing Python code in order to improve its readability and consistency. It includes guidance on things such as:

- **Naming conventions**, e.g. use lower case words separated by underscores for function names
- **Code layout**, e.g. leave blank lines around function definitions; maximum line length should be 79 characters
- **Indentation**, e.g. use 4 spaces rather than a tab
- **Comments**, e.g. comment your code
- **White space in expressions**, e.g. only add whitespace around the operators with lowest priority

The full document can be found here.

1.4 In computing / computer science what is a program?

In computing, a **program** is a set of unambiguous, ordered instructions that a computer can execute in order to complete a specific task. A program is usually written by a computer programmer in a computer programming language. This is a human-readable language that is compiled into one the computer understands or is executed with the aid of an interpreter (as Python is).

1.5 In computing / computer science what is a process?

A **process** (also known as a job) is the instance of a computer program that the computer is running (i.e. it is a program in execution). It consists of the program

loaded into memory, along with the resources that it needs to operate. The operating system (OS) allocates resources to a process. This is known as **scheduling**.

A process can be in one of three states:

- Waiting—where it is waiting for a resource to become available
- Ready—where it has what it needs to run but is not currently active in the CPU
- Running—where it is currently running within the CPU

There can be tens to hundreds of processes running on your computer at any time. You can see all the processes being run by your computer from the Activity Monitor (Mac OS) or Task Manager (Windows). A process runs independently and is isolated from other processes. This means that if a problem arises in one process, it shouldn't affect any others.

1.6 In computing / computer science what is cache?

Cache is a type of high-speed computer memory that is either part of the central processing unit (CPU) or sits between the CPU and main/random access memory (RAM) depending on the type of cache. It only has a small capacity, ranging from a few KB to a few MB. It is designed to temporarily store instructions and data that the CPU is likely to reuse. The CPU will check with the cache before the RAM, which is the main place for storing instructions and data while a process is being executed.

1.7 In computing / computer science what is a thread and what do we mean by multithreading?

A **thread** (short for thread of execution) is a unit of execution within a process. It is the smallest unit of programmed instructions that resources can be allocated to by the scheduler.

A process can consist of one thread (**single-threaded**) or numerous threads (**multithreaded**). In a multithreaded process, threads will share most of their resources and can communicate easily with each other. This makes them vulnerable to problems caused by other threads. An example of a multithreaded process could be when you have multiple tabs open in a web browser, with each tab operating on a different thread.

1.8 In computing / computer science what is concurrency and parallelism and what are the differences?

Parallelism is when multiple processes or threads are executed simultaneously. This is possible on a system with multiple processors (which includes most modern systems). It is not possible for multiple processes or threads to execute at the same time on a single processor. In this situation, the scheduler will allocate CPU time to each process or thread such that tasks are run one at a time. This is known as **concurrency**. In reality, the switching back and forth between tasks happens so fast that it is not noticeable.

1.9 What is GIL in Python and how does it work?

The **global interpreter lock (GIL)** is a lock that allows only one thread be executed at once, even in a multithreaded process on a machine with more than one CPU.

Python objects have a **reference count**, which keeps track of how many times the object is pointed to. If the count reaches zero, the memory used to store the object is freed up for use. If two or more threads were to operate at the same time, they might all change the reference count of the object, increasing or decreasing its amount simultaneously. This could mean that the reference count reaches zero in one thread, resulting in the memory being released, only for it to still be required by another thread. It could also mean that the reference count never reaches zero, even if it should have done, resulting in leaked memory.

Locks can be put on data structures that are required by multiple threads but this can result in multiple locks and deadlocks. Instead, the GIL works by putting a single lock on the Python interpreter and requiring any code to acquire the interpreter lock in order to execute.

1.10 What do these software development principles mean: DRY, KISS, BDUF

- **Do not repeat yourself (DRY)** – Reduce repetition by avoiding writing the same code or logic over and over again. Write code in reusable blocks that can be called again later.
- **Keep it simple stupid (KISS)** – Keep code clear and simple so that it is easy to understand.
- **Big design upfront (BDUF)** – A way of developing software such as the whole design solution is completed and perfected before it is implemented. Found in the waterfall development method rather than in agile software development.

1.11 What is a Garbage Collector in Python and how does it work?

The **garbage collector** is part of Python's automatic memory management, releasing memory when it is no longer required. It works by keeping a record of how many times an object is referenced during the execution of a program. If the reference count reaches zero, the memory that was used to store the object is deallocated.

1.12 How is memory managed in Python?

Memory is managed by the **Python memory manager**. As explained in the previous answer, this keeps a reference count of Python objects and when it reaches zero, the memory used to store the object is deallocated. This is all done by the Python

memory manager for you so you don't have to keep a track of the memory allocation yourself. Python objects and data structures are stored in a private heap. Memory is dynamically allocated within the private heap at runtime (as opposed to being statically allocated to a stack at compile time). The use of dynamically allocated memory allows memory to be freed and reused.

1.13 What is a Python module?

A **module** is a Python file (.py) containing pre-written code that can be imported into another Python script and used there without it having to be written out in full. The code contains definitions of functions, classes and variables. It can also contain executable statements.

You can create your own modules. This is often what you would do if you had a long program. Splitting it into multiple shorter scripts that can be imported into each other makes the code easier to read and maintain.

You can also use modules published by others. These are often published alongside other related modules in a **package**.

1.14 What is docstring in Python?

A docstring is written text (a literal string) that documents a method, function, class or module. It is usually provided before the code to explain what it does and how it should be used. Docstrings can span multiple lines but the first should always be a summary statement. It is convention to denote a docstring with three double quotes:

```
"""This is a docstring"""
```

1.15 What is pickling and unpickling in Python? Example usage.

Pickling is the process of converting a Python object structure to a stream of characters (byte stream) that contains all of the information required to reconstruct the object in another Python script. **Unpickling** is the reconstruction of the object. In other words, pickling and unpickling allow you to serialise and de-serialise a Python object structure. The process requires the `pickle` module.

Pickling is used to store a Python object in a file/database, to save the program state across sessions so that it can pick up at the same place when restarted (persistence) or to send data over a network (marshalling).

1.16 What are the tools that help to find bugs or perform static analysis?

The `pdb` module – This is the standard Python debugger.

Built-in IDE tools – You can run a program in debugging mode in an IDE like Pycharm.

1.17 How are arguments passed in Python by value or by reference? Give an example.

Pass by reference – passing an argument to a function that is a reference to a variable that already exists in memory, rather than an independent copy of that variable.

Pass by value – passing an argument to a function that is an independent value. The argument passed becomes an independent copy of the original.

Python does a combination of both of these, which is often referred to as **pass by assignment**.

All data in Python are stored in memory as **objects** that each have a **unique ID**, a **type**, a **value consistent with its type** and a **reference count**. When an object is assigned to a variable, Python creates the object first. It then creates the variable, which simply points to the object. Numerous variables can all point to the same object.

When an immutable-type object (e.g. string, int, tuple etc.) is assigned to a variable, any changes applied to the variable will not result in a change to the original object. **Immutable-type objects are therefore passed to functions by value**. In the following example, `a` is passed as `self` to the `upper()` method by value and a new object – `'CODE FIRST GIRLS'` – is temporarily created. As there is nothing pointing to it, its reference count is zero so is removed from memory. The object that `a` points to remains unchanged.

```
>>> a = 'code first girls'
>>> a.upper()
'CODE FIRST GIRLS'
>>> a
'code first girls'
```

However, when a mutable-type object (e.g. list, dict, set) is assigned to a variable, any changes applied to the variable will result in a change to the original object. **Mutable-type objects are therefore passed to functions by reference**. In the following example, `b` is passed as `self` to the `append()` method. No new object is created, the method modifies the original object that `b` points to.

```
>>> b = ['code', 'first']
>>> b.append('girls')
>>> b
['code', 'first', 'girls']
```

1.18 What are Dictionary and List comprehensions in Python? Provide examples.

Mutable types (lists, dictionaries and sets) have **comprehensions**. A comprehension is a way of creating a list, dictionary or set. Although there are

other ways to create these objects, it is more Pythonic and faster to use a comprehension.

To create a list, the syntax is:

```
[expression for item in iterable if condition]
```

Note that the `if` condition is optional. The following example creates a list of multiples of 5:

```
>>> list_of_numbs = [number * 5 for number in range(1, 5)]
>>> list_of_numbs
[5, 10, 15, 20]
```

To create a dictionary, the syntax is:

```
{key_expression: value_expression for expression in iterable if condition}
```

Note that the `if` condition is again optional. The following example creates a dictionary of the number of occurrences of each letter in a string using dictionary comprehension:

```
>>> s = 'code first girls'
>>> letter_occurrence = {letter: s.count(letter) for letter in s}
{'c': 1, 'o': 1, 'd': 1, 'e': 1, ' ': 2, 'f': 1, 'i': 2, 'r': 2,
's': 2, 't': 1, 'g': 1, 'l': 1}
```

1.19 What is namespace in Python?

As explained in question 17, when an object is assigned to a variable, the object is created first and then a variable is created that points to that object. The variable is a name that can be used to reference that object. Each object is unique and has its own unique ID. Numerous variables (i.e. names) can point to the same object. Names are not limited simply to objects that have been explicitly assigned to variable. For example, a function name is a name that points to a function object.

A **namespace** is a collection of all the names that reference Python objects. Different namespaces can all exist at the same time. They are all isolated from each other. The **built-in namespace** is created when the Python interpreter is started. Each module then has its own **global namespace** containing the names

that are defined at the level of the main program. Each function has its own **local namespace** that is created when the function is run and remains in existence until it terminates. When a function calls another function within it, this is said to be an enclosing function and has an **enclosing namespace**. The enclosed function inside it has a local namespace.

1.20 What is pass in Python?

It is not possible to run or create a loop, function definition, class definition or if statement with empty code. It would raise an exception. **Pass** is a null statement that is used as a placeholder for future code. It is not ignored as it mitigates an exception being raised, however, nothing happens when a pass statement is executed.

1.21 What is unit test in Python?

A **unit test** is a program that is written to test the logic in another program you have written. It independently evaluates small chunks of the program (such as a function) to ensure that each returns the desired output. Where the function under test requires a value generated externally to the function (e.g. an input, another function, data from an API etc.), **a mock object** can be supplied. This ensures that it is only the code itself under test and increases code coverage i.e. the portion of the code that is able to be tested.

1.22 In Python what is slicing?

Slicing is a way of extracting a sub-part of any sequential data such as a substring from a string or a subsequence of items from a list. It can be done simply with brackets, the syntax of which is:

```
sequence[start:stop:step]
```

`start` and `stop` are offsets and refer to index positions. It is easiest to think of the index positions as being between the characters/items. Index 0 is before the

first character/item. `start` defaults to `0` and stop defaults to `len(sequence)`. *Note that you are specifying the part of the subsequence that you want to keep! Not the part you want to cut off!*

```
>>> s = 'abcdefghijklmnopqrstuvwxyz'

>>> s[5:]
'fghijklmnopqrstuvwxyz'

>>> s[:3]
'abc'

>>> s[14:18]
'opqr'

>>> s[3:10:2]
'dfhj'
```

1.23 What is a negative index in Python?

Positive indexes count from the start of the sequence, while **negative indexes** count from the end of the sequence. The index between the last two characters/item is –1. The numbers go down in value (i.e. –2, –3, –4 etc.) towards the left. Here are some examples:

```
>>> s = 'abcdefghijklmnopqrstuvwxyz'

>>> s[-1]
'z'

>>> s[−5:]
'vwxyz'

>>> s[−15:−10]
'lmnop'
```

1.24 How can the ternary operators be used in python? Give an example.

Ternary operators are a type of conditional statement that evaluates something based on whether it is true or false. Normally conditional statements span multiple lines, however, a ternary operator can span just one line. The syntax is:

```
[if_true] if [expression] else [if_false]
```

The following demonstrates it in use:

```
>>> age = 20
>>> is_teenager = True if 13 <= age <= 19 else False
>>> is_teenager
False

>>> spent = 48.50
>>> discount = 5 if spent > 50 else 0
>>> discount
0
```

1.25 What does this mean: *args, **kwargs? And why would we use it?

Both are used to pass a variable number of arguments to a function. You would use them when you write a function definition but know that the number of arguments that will be passed to the function could vary with each function call.

***args** - used to pass non-keyword arguments

****kwargs** - used to pass keyword arguments

Here is an example:

```
>>> def add_numbers(*num):
        sum = 0
        for n in num:
            sum += n
        return sum

>>> add_numbers(2, 6, 4, 7)
19

>>> add_numbers(3, 2)
5
```

1.26 How are range and xrange different from one another?

Both functions provide a way of generating a sequence of integers within a given range.

In Python 2, the `range()` function creates a list of integers, which is limited by the space available in memory and can be a problem if you are working with very

large ranges. The Python 2 `xrange()` function is a **generator function** that creates a sequence object that **evaluates lazily**. This means that it does not store every number in the range in memory and only computes the next value when it is required, which makes it useful if you are dealing with very large ranges.

Python 3 does not have a function named `xrange`. The `range()` function in Python 3 works like the Python 2 `xrange()` function.

1.27 What is Flask and what can we use it for?

Flask is a **web development framework** for Python. A framework is a set of tools, libraries and resources that makes it easier to build a web application. You would use it to build web applications. Flask can be installed from PyPI using pip.

1.28 What are clustered and non-clustered index in a relational database?

When a record is entered into a table, the server does not try to place it in any particular location, it just gets put in the next available location. This means that we cannot derive any meaning from the order of the records in a table. When a table is queried, the server must search through all rows in the table to find the right ones. This is known as a table scan. If the table is very large it will take a long time for the server to search through all the records. Indexing speeds up this process, improving query efficiency.

A **clustered index** sorts the rows of data in a table based on their values in a specified column and stores the actual table in this order. There can only be one clustered index for a table. This is usually the primary key. In MySQL, the server automatically creates a clustered index from the primary key. A table with a clustered index is called a clustered table.

A **non-clustered (secondary) index** stores a specific column in a special table. The column is stored in a sorted order alongside values describing where the rows are physically located in the actual table.

1.29 What is a ‘deadlock’ a relational database?

A **deadlock** in a relational database is when two or more transactions are waiting for each other to give up locks. All activity comes to a standstill until the deadlock is detected by the DBMS and at least one of the transactions is aborted.

1.30 What is a ‘livelock’ a relational database?

A **livelock** is when a request for an exclusive (write) lock is repeatedly denied because there are multiple shared (read) locks. If two processes are each holding a read lock on a resource but need a write lock on the other's resource to continue executing, this would be a livelock. Neither process could progress as you cannot obtain a write lock on a resource if there is a read lock on it.

2 | Python string methods (29 points)

Describe each method and provide an example.

Note that strings are immutable so where a string is returned in all of the following, this is a new object and the original string itself has not been changed.

`[]` denotes optional arguments.

Method	Description	Syntax	Example

`capitalize()`	Capitalises the first letter of a string.	`str.capitalize()`	`>>> s = 'code first girls'`
			`>>> s.capitalize()`
			`'Code first girls'`

`casefold()`	Converts string to lower case, including caseless letters, which the `lower()` method would ignore.	`str.casefold()`	`>>> s = 'außen'`
			`>>> s.casefold()`
			`'aussen'`

`center()`	Returns a string padded with the specified character. This character is an optional argument and if it is not provided, it will use the default character of a space. This effectively centre-aligns the string. A width variable must be provided. This is the total string length required including the string and the padding.	`str.center(width, [fillchar])`	`>>> s = 'code first girls'`
			`>>> s.center(26)`
			`'     code first girls     '`
			
			`>>> s.center(22, '*')`
			`'***code first girls***'`

`count()`	Tells you how many times a substring occurs within a string. You can provide start and end arguments to dictate which characters in the string is to be searched.	`str.count(substring, start, end)`	`>>> s = 'code first girls'`
			`>>> s.count('i')`
			`2`
			
			`>>> s.count('l', 0, 5)`
			`2`

`endswith()`	Returns true or false depending on whether the string ends with the specified character. You can provide start and end arguments to dictate which portion of the string you want the method to work on if not the whole string.	`str.endswith(suffix[, start[, end]])`	`>>> s = 'code first girls'`
			`>>> s.endswith('o')`
			`False`
			
			`>>> s.endswith('t', 5, 10)`
			`True`

Method	Description	Syntax	Example

`find()`	Returns the positional index of the first occurrence of the first character of the specified substring in the string. Can specify a portion of the string to search through with start and end arguments. It can be used even if you are not sure if the substring is present in the string.	`str.find(substring[, start[, end]])`	`>>> s = 'code first girls'`
			`>>> s.find('rs')`
			`7`

`format()`	Offers many ways to visually improve the output of a string containing placeholders.	`str.format(value1, value2, …)`	`>>> item = 'bread'`
			`>>> '{} added to basket'.format(item.capitalize())`
			`'Bread added to basket'`
			
			`>>> price = 1.30`
			`>>> 'The cost is: £{:.2f}'.format(price)`
			`'The cost is: £1.30'`

`index()`	Similar to the `find()` method but should only be used if you are sure the substring is present in the string. Raises an exception if substring is not found.	`str.index(substring[, start[, end]])`	`>>> s = 'code first girls'`
			`>>> s.index('e')`
			`3`

`isalnum()`	Checks if the string is all alphanumeric characters (letters and numbers only). Returns `True` or `False`. Note: whitespaces are NOT alphanumeric.	`str.isalnum()`	`>>> s = 'Codefirstgirls3'`
			`>>> s.isalnum()`
			`True`
			
			`>>> s = 'code first girls5!'`
			`>>> s.isalnum()`
			`False`

`isalpha()`	Checks if the string contains only alphabets (letters) . Returns `True` or `False`.	`str.isalpha()`	`>>> s = 'codefirstgirls'`
			`>>> s.isalpha()`
			`True`
			
			`>>> s = 'code first girls'`
			`>>> s.isalpha()`
			`False`

`isdigit()`	Checks if the string contains only digits. These include integers, decimals, subscript numbers and superscript numbers. Returns `True` or `False`.	`str.isdigit()`	`>>> s = '14848293'`
			`>>> s.isdigit()`
			`True`
			
			`>>> s = 'Code First Girls 3'`
			`>>> s.isdigit()`
			`False`

Method	Description	Syntax	Example

`islower()`	Checks if string is all lower case. Returns `True` or `False`.	`str.islower()`	`>>> s = 'code first girls'`
			`>> s.islower()`
			`True`

`isnumeric()`	Checks if the string contains only numeric characters. They can be integers, decimals, subscript, superscript, fractions. Returns `True` or `False`.	`str.isnumeric()`	`>>> s = '35'`
			`>>> s.isnumeric()`
			`True`

`isspace()`	Checks if the string contains only spaces . Returns `True` or `False`.	`str.isspace()`	`>>> s = '  '` `True`

`istitle()`	Checks if the string is in title case. Returns `True` or `False`.	`str.istitle()`	`>>> s = 'Code First Girls'`
			`>>> s.istitle()`
			`True`

`isupper()`	Checks if the string is all in upper case . Returns `True` or `False`.	`str.isupper()`	`>>> s = 'CODE FIRST GIRLS'`
			`>>> s.isupper()`
			`True`

`join()`	The string join method merges strings together so that self (str1) separates each element of the string provided as an argument (str2).	`str1.join(str2)`	`>>> s1 = '-'`
			`>>> s2 = 'code first girls'`
			`>>> s1.join(s2)`
			`'c-o-d-e-f-i-r-s-t-g-i-r-l-s'`
			
			`>>> s1 = '123456'`
			`>>> s2 = '.'`
			`>>> s2.join(s1)`
			`1.2.3.4.5.6`

`lower()`	Puts all letters in a string into lower case.	`str.lower()`	`>>> s = 'Code First Girls'`
			`>>> s.lower()`
			`'code first girls'`

`lstrip()`	Returns a string with leading characters removed. Provided characters must match the first characters of the string or nothing will happen. You can choose how many you want to strip.	`str.lstrip([chars])`	`>>> s = 'code first girls'`
			`>>> s.lstrip('code ')`
			`'first girls'`

`replace()`	Returns a string that has been replaced by another string. You can specifiy an optional count, which is the number of times you want to replace the old string with the new string.	`str.replace(old, new[, count])`	`>>> s = 'code first girls'`
			`>>> s.replace('first', '*****')`
			`'code ***** girls'`

`rsplit()`	Same as `split()` but starts splitting from the right. The difference is noticeable when maxsplit is specified. Compare the example here with the second example given for `split()`.	`str.rsplit([separator[, maxsplit]])`	`>>> s = 'code.first.girls'`
			`>>> s.rsplit('.', 1)`
			`['code.first', 'girls']`

Method	Description	Syntax	Example

`rstrip()`	Same as `strip()` but removes characters from the right of the string only.	`str.rstrip([chars])`	`>>> s = ' code first girls    '`
			`>>> s.rstrip()`
			`' code first girls'`

`split()`	Splits string into a list of strings. If separator is not specified, it will use white space as a separator. The separator marks the point from which to start splitting (from the left).	`str.split([separator[, maxsplit]])`	`>>> s = 'code first girls'`
			`>>> s.split()`
			`['code', 'first', 'girls']`
			
			`>>> s = 'code.first.girls'`
			`>>> s.split('.', 1)`
			`['code', 'first.girls']`

`splitlines()`	Splits string at line breaks and returns each line as an item in a list. It recognises a range of end-of-line escape characters, e.g. `\n` and `\r`. If `keepends` is set to `True`, the end-of-line escape characters will be shown in the output.	`str.splitlines([keepends])`	`>>> s = 'code\nfirst girls'`
			`>>> s.splitlines()`
			`['code', first girls']`
			
			`>>> s.splitlines(True)`
			`['code\n', 'first girls']`

`startswith()`	Checks if the string starts with the specified prefix. Can specify a portion of the string to look at. Returns `True` or `False`.	`str.startswith(prefix[, start[, end]])`	`>>> s = 'code first girls'`
			`>>> s.startswith('co')`
			`True`

`strip()`	Removes leading and trailing characters. When characters aren't specified, it removes leading and trailing white space.	`str.strip([chars])`	`>>> s = '  code first girls  '`
			`>>> s.strip()`
			`'code first girls'`
			
			`>>> s = '*code first girls*****'`
			`>>> s.strip('*')`
			`'code first girls'`

`swapcase()`	Swaps the case of letters to the opposite one, i.e. lower to upper and upper to lower.	`str.swapcase()`	`>>> s = 'Code First Girls'`
			`>>> s.swapcase()`
			`'cODE fIRST gIRLS'`

`title()`	Capitalises first letter of each word in a string (title case).	`str.title()`	`>>> s = 'hello world'`
			`>>> s.title()`
			`'Hello World'`

`upper()`	Capitalises all letters in a string.	`str.upper()`	`>>> s = 'hello world'`
			`>>> s.upper()`
			`'HELLO WORLD'`

3 | Python list methods (11 points)

Describe each method and provide an example.

Method	Description	Syntax	Example

`append()`	Add a new item at the end of a list.	`list.append(item)`	`>>> l = [1, 2, 3]`
			`>>> l.append(4)`
			`>>> l`
			`[1, 2, 3, 4]`

`clear()`	Remove all items from a list.	`list.clear()`	`>>> l = [1, 2, 3]`
			`>>> l.clear()`
			`>>> l`
			`[]`

`copy()`	Creates a copy of a list that is not changed when the old list is modified. The old list won't change when the copy is modified.	`list.copy()`	`# Not using copy()`
			`>>> l1 = [1, 2, 3]`
			`>>> l2 = l1`
			`>>> l1.append(4)  # modify l1`
			`>>> l1  # l1 is modified`
			`[1, 2, 3 ,4]`
			`>>> l2  # l2 is also modified`
			`[1, 2, 3, 4]`
			
			`# Using copy()`
			`>>> l1 = [1, 2, 3]`
			`>>> l2 = l1.copy()`
			`>>> l1.append(4)  # modify l1`
			`>>> l1. # l1 is modified`
			`[1, 2, 3, 4]`
			`>>> l2  # l2 is not modified`
			`[1, 2, 3]`

`count()`	Counts the number of times a specified item occurs in a list.	`list.count(item)`	`>>> l = [1, 2, 2, 55, 1, 2]`
			`>>> l.count(2)`
			`3`

Method	Description	Syntax	Example

`extend()`	Adds multiple items to the end of a list. The items can be provided as any type of iterable (list, dict, string, tuple etc.	`list.extend(iterable)`	`>>> l1 = [1, 2, 3]`
			`>>> l2 = [4, 5, 6]`
			`>>> l1.extend(l2)`
			`>>>l1`
			`[1, 2, 3, 4, 5, 6]`

`index()`	Returns the index of a specified item within a list.	`list.index(item[, start, end])`	`>>> l = ['cat', 'dog', 'bee']`
			`>>> l.index('bee')`
			`2`

`insert()`	Inserts an item into a list at the specified index.	`list.insert(index, item)`	`>>> l = ['cat', 'dog', 'bee']`
			`>>> l.insert(1, 'fish')`
			`>>> l`
			`['cat', 'fish', 'dog', 'bee']`

`pop()`	The item at a given index in the list is removed from the list and returned.	`list.pop(index)`	`>>> l = ['cat', 'dog', 'bee']`
			`>>> l.pop(2)`
			`'bee'`
			`>>> l`
			`['cat', 'dog']`

`remove()`	Removes the first occurrence of the specified item in a list.	`list.remove(item)`	`>>> l = ['cat', 'dog', 'bee', 'dog']`
			`>>> l.remove('dog')`
			`>>> l`
			`['cat', 'bee', 'dog']`

`reverse()`	Reverse the items in a list.	`list.reverse()`	`>>> l = ['cat', 'dog', 'bee', 'dog']`
			`>>> l.reverse(l)`
			`>>> l`
			`['dog', 'bee', 'dog', 'cat']`

`sort()`	Sorts the items in a list by ascending order. Will sort by descending order if `reverse = True`. You can specify a key to sort by a specific function, e.g. `len`.	`list.sort([key, reverse])`	`>>> l = ['cat', 'dog', 'bee', 'dog']`
			`>>> l.sort()`
			`>>> l`
			`['bee', 'cat', 'dog', 'dog']`

4 | Python tuple methods (2 points)

Describe each method and provide an example.

Method	Description	Syntax	Example

`count()`	Returns the number of occurrences of a specified element within a tuple.	`tuple.count(element)`	`>>> t = ('h', 'e', 'l', 'l', 'o')`
			`>>> t.count('l')`
			`2`

`index()`	Returns the index of the first occurrence of a specified element within a tuple. Start and end can be specified to search between specified indexes.	`tuple.index(element, start, end)`	`>>> t.index('e')` `1`

5 | Python dictionary methods (11 points)

Describe each method and provide an example.

Method	Description	Syntax	Example

`clear()`	Remove all items from a dictionary.	`dict.clear()`	`>>> d = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d.clear()`
			`>>> d`
			`{}`

`copy()`	Copies a dictionary so that modifications to the original dictionary would not modify the new dictionary.	`dict.copy()`	`>>> d1 = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d2 = d1.copy()`
			`>>> d2`
			`{'item': 'armchair', 'colour': 'green',`
			`'price': 149.99}`

`fromkeys()`	Create a new dictionary. `sequence` specifies the keys for the new dictionary from the one that the method is being applied to. `value` specifies the value that will be added to all of the new keys.	`dict.fromkeys(sequence[, value])`	`>>> d = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d.fromkeys(d)`
			`{'item': None, 'colour': None, 'price':`
			`None}`

`get()`	Returns the value for the specified key. If key is not found, it can return `value`.	`dict.get(key[, value])`	`>>> d = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d.get('price')`
			`149.99`

Method	Description	Syntax	Example

`items()`	Returns a view object containing a list of key-value pairs.	`dict.items()`	`>>> d = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d.items()`
			`dict_items([('item', 'armchair'), ('colour', 'green'), ('price', 149.99)])`

`keys()`	Returns a view object containing a list of keys.	`dict.keys()`	`>>> d = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d.keys()`
			`dict_keys(['item', 'colour', 'price'])`

`pop()`	Returns an element and removes it from the dictionary.	`dict.pop(key[, default])`	`>>> d = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> key.pop('colour')`
			`'green'`
			`>>> d`
			`{'item': 'armchair', 'price': 149.99}`

`popitem()`	Returns the last element and removes it from the dictionary.	`dict.popitem()`	`>>> d = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d.popitem()`
			`('price', 149.99)`
			`>>> d`
			`{'item': 'armchair', 'colour': 'green'}`

Method	Description	Syntax	Example

`setdefault()`	Returns the value of the specified key or inserts it with the specified default value if it does not exist.	`dict.setdefault(key[, default_value])`	`>>> d = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d.setdefault('no_in_stock', 0)`
			`0`
			`>>> d`
			`{'item': 'armchair', 'colour': 'green',`
			`'no_in_stock': 0}`

`update()`	Inserts the specified items into the dictionary.	`dict.update([iterable])`	`>>> d1 = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d2 = {`
			`'no_in_stock': 5,`
			`'no_on_order': 3`
			`}`
			`>>> d1.update(d2)`
			`{'item': 'armchair', 'colour': 'green',`
			`'price': 149.99, 'no_in_stock': 5,`
			`'no_on_order': 3}`

`values()`	Returns a view object containing a list of values.	`dict.values()`	`>>> d = {`
			`'item': 'armchair',`
			`'colour': 'green',`
			`'price': 149.99`
			`}`
			`>>> d.values()`
			`dict_values(['armchair', 'green', 149.99])`

6 | Python set methods (12 points)

Describe each method and provide an example.

Method	Description	Syntax	Example

`add()`	Adds an item to the set.	`set.add(element)`	`>>> s = {'cat', 'dog', 'bee'}`
			`>>> s.add('fish')`
			`>>> s`
			`{'fish', 'dog', 'cat', 'bee'}`

`clear()`	Removes all items from the set.	`set.clear()`	`>>> s = {'cat', 'dog', 'bee'}`
			`>>> s.clear()`
			`>>> s`
			`set()`

`copy()`	Copies a set so that modifications to the original set would not modify the new set.	`set.copy()`	`>>> s1 = {'cat', 'dog', 'bee'}`
			`>>> s2 = s1.copy()`
			`s2`
			`{'dog', 'cat', 'bee'}`

`difference()`	Returns a set that contains the difference between two sets. The items that are kept are those in the set that the method is being applied to, not the one passed as an argument.	`set.difference(set)`	`>>> s1 = {'cat', 'dog', 'bee'}`
			`>>> s2 = {'sheep', 'dog', 'cow'}`
			`>>> s1.difference(s2)`
			`{'cat', 'bee'}`

`intersection()`	Returns only the items that are present in all given sets.	`set.intersection(set1[, set2, …, setn])`	`>>> s1 = {'cat', 'dog', 'bee'}`
			`>>> s2 = {'sheep', 'dog', 'cow'}`
			`>>> s1.intersection(s2)`
			`{'dog'}`

`issubset()`	Checks if the set that the method is being applied to occurs in the set provided as an argument. Returns `True` or `False`.	`set.issubset(set)`	`>>> s1 = {'cat', 'dog'}`
			`>>> s2 = {'sheep', 'dog', 'cat'}`
			`>>> s1.issubset(s2)`
			`True`

`issuperset()`	Checks if the set that the method is being applied to is a superset of the set provided as an argument. Returns `True` or `False`.	`set.issuperset(set)`	`>>> s1 = {'sheep', 'dog', 'cat'}`
			`>>> s2 = {'cat', 'sheep'}`
			`>>> s1.issuperset(s2)`
			`True`

Method	Description	Syntax	Example

`pop()`	Removes a random item from the set.	`set.pop()`	`>>> s = {'cat', 'dog', 'bee'}`
			`>>> s.pop()`
			`'dog'`
			`>>> s`
			`{'cat', 'bee'}`

`remove()`	Removes the specified item from the set.	`set.remove(item)`	`>>> s = {'cat', 'dog', 'bee'}`
			`>>> s.remove('bee')`
			`>>> s`
			`{'dog', 'cat'}`

`symmetric_ difference()`	Returns a set that contains the items from both sets, except those that are present in both.	`set.symmetric_difference(set)`	`>>> s1 = {'cat', 'dog', 'bee'}`
			`>>> s2 = {'sheep', 'dog', 'cow'}`
			`>>> s1.symmetric_difference(s2)`
			`{'bee', 'sheep', 'cow', 'cat'}`

`union()`	Returns a set that contains all items from all sets.	`set.union(set1[, set2, …, setn])`	`>>> s1 = {'cat', 'dog', 'bee'}`
			`>>> s2 = {'sheep', 'dog', 'cow'}`
			`>>> s1.union(s2)`
			`{'sheep', 'bee', 'cow', 'dog', 'cat'}`

`update()`	Update the set that the method is being applied to by adding items form the set provided as an argument.	`set.update(set)`	`>>> s1 = {'cat', 'dog', 'bee'}`
			`>>> s2 = {'sheep', 'dog', 'cow'}`
			`>>> s1.update(s2)`
			`{'sheep', 'bee', 'cow', 'dog', 'cat'}`

7 | Python file methods (11 points)

Describe each method and provide an example.

Method	Description	Syntax	Example

`read()`	Returns the contents of a file. If you provide an integer as an argument, it will return that many bytes (characters).	`file.read(size)`	`>>> f = open('myfile.txt', 'r')`
			`>>> f.read()`
			`'Hello,\nThis is my test file.'`
			`>>> f.close()`

`readline()`	Returns one line of a file. If `size` is specified, it will return that many number of bytes (i.e. number of characters) from the line. The default is to return the whole line.	`file.readline(size)`	`>>> f = open('myfile.txt', 'r')`
			`>>> f.readline()`
			`'Hello,\n'`
			`>>> f.close()`

`readlines()`	Returns a list of lines. `hint` will limit the number of lines returned.	`file.readlines(hint)`	`>>> f = open('myfile.txt', 'r')`
			`>>> f.readline()`
			`['Hello,\n', 'This is my test file.']`
			`>>> f.close()`

`write()`	Writes the given text or byte object (i.e. the `byte` argument) to a file.	`file.write(byte)`	`>>> f = open('newfile.txt', 'w')`
			`>>> f.write('Adding some text!\n')`
			`>>> f.close()`
			
			`>>> f = open('newfile.txt', 'r')`
			`>>> f.read()`
			`'Adding some text!\n'`
			`>>> f.close()`

`writelines()`	Write the list of strings provided to a file. Note that you will need to manually provide end of line breaks to force each item onto a separate line.	`file.writelines(list)`	`>>> f = open('newfile.txt', 'a')`
			`>>> f.writelines(['one\n', 'two\n'])`
			`>>> f.close()`
			
			`>>> f = open('newfile.txt', 'r')`
			`>>> f.read()`
			`'Adding some text!\none\ntwo\n'`
			`>>> f.close()`