1. First Programming Concepts Cheat Sheet
2. **Functions and Keywords**

Functions and keywords are the building blocks of a language’s syntax.

Functions are pieces of code that perform a unit of work. In the examples we've seen so far, we've only encountered the print() function, which prints a message to the screen. We'll learn about a lot of other functions in later lessons but, if you're too curious to wait until then, you can discover all the functions available [here](https://docs.python.org/3/library/functions.html).

Keywords are reserved words that are used to construct instructions. We briefly encountered for and in in our first Python example, and we'll use a bunch of other keywords as we go through the course. For reference, these are all the reserved keywords:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass |  |
| break | except | in | raise |  |

You don't need to learn this list; we'll dive into each keyword as we encounter them. In the meantime, you can see examples of keyword usage [here](https://www.programiz.com/python-programming/keyword-list).

1. **Arithmetic operators**

Python can operate with numbers using the usual mathematical operators, and some special operators, too. These are all of them (we'll explore the last two in later videos).

* **a + b**= Adds a and b
* **a - b** = Subtracts b from a
* **a \* b** = Multiplies a and b
* **a / b** = Divides a by b
* **a \*\* b** = Elevates a to the power of b. For non integer values of b, this becomes a root (i.e. a\*\*(1/2) is the square root of a)
* **a // b** = The integer part of the integer division of a by b
* **a % b** = The remainder part of the integer division of a by b

# Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

|  |  | **Built-in Functions** |  |  |
| --- | --- | --- | --- | --- |
| [abs()](https://docs.python.org/3/library/functions.html#abs) | [delattr()](https://docs.python.org/3/library/functions.html#delattr) | [hash()](https://docs.python.org/3/library/functions.html#hash) | [memoryview()](https://docs.python.org/3/library/functions.html#func-memoryview) | [set()](https://docs.python.org/3/library/functions.html#func-set) |
| [all()](https://docs.python.org/3/library/functions.html#all) | [dict()](https://docs.python.org/3/library/functions.html#func-dict) | [help()](https://docs.python.org/3/library/functions.html#help) | [min()](https://docs.python.org/3/library/functions.html#min) | [setattr()](https://docs.python.org/3/library/functions.html#setattr) |
| [any()](https://docs.python.org/3/library/functions.html#any) | [dir()](https://docs.python.org/3/library/functions.html#dir) | [hex()](https://docs.python.org/3/library/functions.html#hex) | [next()](https://docs.python.org/3/library/functions.html#next) | [slice()](https://docs.python.org/3/library/functions.html#slice) |
| [ascii()](https://docs.python.org/3/library/functions.html#ascii) | [divmod()](https://docs.python.org/3/library/functions.html#divmod) | [id()](https://docs.python.org/3/library/functions.html#id) | [object()](https://docs.python.org/3/library/functions.html#object) | [sorted()](https://docs.python.org/3/library/functions.html#sorted) |
| [bin()](https://docs.python.org/3/library/functions.html#bin) | [enumerate()](https://docs.python.org/3/library/functions.html#enumerate) | [input()](https://docs.python.org/3/library/functions.html#input) | [oct()](https://docs.python.org/3/library/functions.html#oct) | [staticmethod()](https://docs.python.org/3/library/functions.html#staticmethod) |
| [bool()](https://docs.python.org/3/library/functions.html#bool) | [eval()](https://docs.python.org/3/library/functions.html#eval) | [int()](https://docs.python.org/3/library/functions.html#int) | [open()](https://docs.python.org/3/library/functions.html#open) | [str()](https://docs.python.org/3/library/functions.html#func-str) |
| [breakpoint()](https://docs.python.org/3/library/functions.html#breakpoint) | [exec()](https://docs.python.org/3/library/functions.html#exec) | [isinstance()](https://docs.python.org/3/library/functions.html#isinstance) | [ord()](https://docs.python.org/3/library/functions.html#ord) | [sum()](https://docs.python.org/3/library/functions.html#sum) |
| [bytearray()](https://docs.python.org/3/library/functions.html#func-bytearray) | [filter()](https://docs.python.org/3/library/functions.html#filter) | [issubclass()](https://docs.python.org/3/library/functions.html#issubclass) | [pow()](https://docs.python.org/3/library/functions.html#pow) | [super()](https://docs.python.org/3/library/functions.html#super) |
| [bytes()](https://docs.python.org/3/library/functions.html#func-bytes) | [float()](https://docs.python.org/3/library/functions.html#float) | [iter()](https://docs.python.org/3/library/functions.html#iter) | [print()](https://docs.python.org/3/library/functions.html#print) | [tuple()](https://docs.python.org/3/library/functions.html#func-tuple) |
| [callable()](https://docs.python.org/3/library/functions.html#callable) | [format()](https://docs.python.org/3/library/functions.html#format) | [len()](https://docs.python.org/3/library/functions.html#len) | [property()](https://docs.python.org/3/library/functions.html#property) | [type()](https://docs.python.org/3/library/functions.html#type) |
| [chr()](https://docs.python.org/3/library/functions.html#chr) | [frozenset()](https://docs.python.org/3/library/functions.html#func-frozenset) | [list()](https://docs.python.org/3/library/functions.html#func-list) | [range()](https://docs.python.org/3/library/functions.html#func-range) | [vars()](https://docs.python.org/3/library/functions.html#vars) |
| [classmethod()](https://docs.python.org/3/library/functions.html#classmethod) | [getattr()](https://docs.python.org/3/library/functions.html#getattr) | [locals()](https://docs.python.org/3/library/functions.html#locals) | [repr()](https://docs.python.org/3/library/functions.html#repr) | [zip()](https://docs.python.org/3/library/functions.html#zip) |
| [compile()](https://docs.python.org/3/library/functions.html#compile) | [globals()](https://docs.python.org/3/library/functions.html#globals) | [map()](https://docs.python.org/3/library/functions.html#map) | [reversed()](https://docs.python.org/3/library/functions.html#reversed) | [\_\_import\_\_()](https://docs.python.org/3/library/functions.html#__import__) |
| [complex()](https://docs.python.org/3/library/functions.html#complex) | [hasattr()](https://docs.python.org/3/library/functions.html#hasattr) | [max()](https://docs.python.org/3/library/functions.html#max) | [round()](https://docs.python.org/3/library/functions.html#round) |  |

1. Data type recap

In Python, text in between quotes -- either single or double quotes -- is a string data type. An integer is a whole number, without a fraction, while a float is a real number that can contain a fractional part. For example, 1, 7, 342 are all integers, while 5.3, 3.14159 and 6.0 are all floats. When attempting to mix incompatible data types, you may encounter a **TypeError**. You can always check the data type of something using the *type()* function.

1. Implicit and Explicit conversion

As we saw earlier in the video, some data types can be mixed and matched due to implicit conversion. Implicit conversion is where the interpreter helps us out and automatically converts one data type into another, without having to explicitly tell it to do so.

By contrast, explicit conversion is where we manually convert from one data type to another by calling the relevant function for the data type we want to convert to. We used this in our video example when we wanted to print a number alongside some text. Before we could do that, we needed to call the *str()* function to convert the number into a string. Once the number was explicitly converted to a string, we could join it with the rest of our textual string and print the result.

1. Function Definition

We looked at a few examples of built-in functions in Python, but being able to define your own functions is incredibly powerful. We start a function definition with the def keyword, followed by the name we want to give our function. After the name, we have the parameters, also called arguments, for the function enclosed in parentheses. A function can have no parameters, or it can have multiple parameters. Parameters allow us to call a function and pass it data, with the data being available inside the function as variables with the same name as the parameters. Lastly, we put a colon at the end of the line.

After the colon, the function body starts. It’s important to note that in Python the function body is delimited by indentation. This means that all code indented to the right following a function definition is part of the function body. The first line that’s no longer indented is the boundary of the function body. It’s up to you how many spaces you use when indenting -- just make sure to be consistent. So if you choose to indent with four spaces, you need to use four spaces everywhere in your code.

Sometimes we don't want a function to simply run and finish. We may want a function to manipulate data we passed it and then return the result to us. This is where the concept of return values comes in handy. We use the return keyword in a function, which tells the function to pass data back. When we call the function, we can store the returned value in a variable. Return values allow our functions to be more flexible and powerful, so they can be reused and called multiple times.

Functions can even return multiple values. Just don't forget to store all returned values in variables! You could also have a function return nothing, in which case the function simply exits.

1. While loop

**(a) Anatomy of a While Loop**

A *while* loop will continuously execute code depending on the value of a condition. It begins with the keyword *while,* followed by a comparison to be evaluated, then a colon. On the next line is the code block to be executed, indented to the right. Similar to an *if* statement, the code in the body will only be executed if the comparison is evaluated to be true. What sets a *while* loop apart, however, is that this code block will keep executing as long as the evaluation statement is true. Once the statement is no longer true, the loop exits and the next line of code will be executed.

**(b) Common Pitfalls With Variable Initialization**

You'll want to watch out for a common mistake: forgetting to initialize variables. If you try to use a variable without first initializing it, you'll run into a **NameError**. This is the Python interpreter catching the mistake and telling you that you’re using an undefined variable. The fix is pretty simple: initialize the variable by assigning the variable a value before you use it.

Another common mistake to watch out for that can be a little trickier to spot is forgetting to initialize variables with the correct value. If you use a variable earlier in your code and then reuse it later in a loop without first setting the value to something you want, your code may wind up doing something you didn't expect. Don't forget to initialize your variables before using them!

**(c) Infinite loops and Code Blocks**

Another easy mistake that can happen when using loops is introducing an infinite loop. An infinite loop means the code block in the loop will continue to execute and never stop. To stop this: (ctrl+c) in terminal and in program use if statement and break statement to break the loop. This can happen when the condition being evaluated in a *while* loop doesn't change. Pay close attention to your variables and what possible values they can take. Think about unexpected values, like zero.

In the Coursera code blocks, you may see an error message that reads "Evaluation took more than 5 seconds to complete." This means that the code encountered an infinite loop, and it timed out after 5 seconds. You should take a closer look at the code and variables to spot where the infinite loop is.

1. For loop

**For Loops Recap**

*For* loops allow you to iterate over a sequence of values. Let's take the example from the beginning of the video:

for x in range(5):

  print(x)

Similar to *if* statements and *while* loops, *for* loops begin with the keyword ***for*** with a colon at the end of the line. Just like in function definitions, *while* loops and *if* statements, the body of the *for* loop begins on the next line and is indented to the right. But what about the stuff in between the *for* keyword and the colon? In our example, we’re using the *range()* function to create a sequence of numbers that our *for* loop can iterate over. In this case, our variable **x** points to the current element in the sequence as the *for* loop iterates over the sequence of numbers. Keep in mind that in Python and many programming languages, a range of numbers will start at 0, and the list of numbers generated will be one less than the provided value. So *range(5)* will generate a sequence of numbers from 0 to 4, for a total of 5 numbers.

Bringing this all together, the range(5) function will create a sequence of numbers from 0 to 4. Our *for* loop will iterate over this sequence of numbers, one at a time, making the numbers accessible via the variable **x** and the code within our loop body will execute for each iteration through the sequence. So for the first loop, **x** will contain 0, the next loop, 1, and so on until it reaches 4. Once the end of the sequence comes up, the loop will exit and the code will continue.

The power of *for* loops comes from the fact that it can iterate over a sequence of any kind of data, not just a range of numbers. You can use *for* loops to iterate over a list of strings, such as usernames or lines in a file.

Not sure whether to use a *for* loop or a *while* loop? Remember that a *while* loop is great for performing an action over and over until a condition has changed. A *for* loop works well when you want to iterate over a sequence of elements.

**A Closer Look at the Range() Function**

Previously we had used the *range()* function by passing it a single parameter, and it generated a sequence of numbers from 0 to one less than we specified. But the *range()* function can do much more than that. We can pass in two parameters: the first specifying our starting point, the second specifying the end point. Don't forget that the sequence generated won't contain the last element; it will stop one before the parameter specified.

The *range()* function can take a third parameter, too. This third parameter lets you  alter the size of each step. So instead of creating a sequence of numbers incremented by 1, you can generate a sequence of numbers that are incremented by 5.

To quickly recap the *range()* function when passing one, two, or three parameters:

* One parameter will create a sequence, one-by-one, from zero to one less than the parameter.
* Two parameters will create a sequence, one-by-one, from the first parameter to one less than the second parameter.
* Three parameters will create a sequence starting with the first parameter and stopping before the second parameter, but this time increasing each step by the third parameter.

1. Loops Cheat Sheet

Check out below for a run down of the syntax for while loops and for loops.

**While Loops**

A while loop executes the body of the loop while the condition remains True.

Syntax:

***while condition:***

***body***

Things to watch out for!

* **Failure to initialize variables.** Make sure all the variables used in the loop’s condition  are initialized before the loop.
* **Unintended infinite loops.** Make sure that the body of the loop modifies the variables used in the condition, so that the loop will eventually end **for all possible values of the variables.**

Typical use:

While loops are mostly used when there’s an unknown number of operations to be performed, and a condition needs to be checked at each iteration.

**For Loops**

A for loop iterates over a sequence of elements, executing the body of the loop for each element in the sequence.

Syntax:

***for variable in sequence***

***body***

The range() function:

range() generates a sequence of integer numbers. It can take one, two, or three parameters:

* range(n): 0, 1, 2, ... n-1
* range(x,y): x, x+1, x+2, ... y-1
* range(p,q,r): p, p+r, p+2r, p+3r, ... q-1 (if it's a valid increment)

**Common pitfalls:**

* **Forgetting that the upper limit of a range() isn’t included.**
* **Iterating over non-sequences.** Integer numbers aren’t iterable. Strings are iterable letter by letter, but that might not be what you want.

Typical use:

For loops are mostly used when there's a pre-defined sequence or range of numbers to iterate.

**Break & Continue**

You can interrupt both while and for loops using the break keyword. We normally do this to interrupt a cycle due to a separate condition.

You can use the continue keyword to skip the current iteration and continue with the next one. This is typically used to jump ahead when some of the elements of the sequence aren’t relevant.

If you want to learn more, check out this [wiki page on for loops](https://wiki.python.org/moin/ForLoop).

1. Recursion

In the past videos, we visited the basic concepts of recursive functions.

A recursive function must include a recursive case and base case. The recursive case calls the function again, with a different value. The base case returns a value without calling the same function.

A recursive function will usually have this structure:

***def recursive\_function(parameters):***

***if base\_case\_condition(parameters):***

***return base\_case\_value***

***recursive\_function(modified\_parameters)***

For more information on recursion, check out these resources:

* [Wikipedia Recursion page](https://en.wikipedia.org/wiki/Recursion)
* See what happens when you [Search Google for Recursion](https://www.google.com/search?q=recursion)

1. String

**(a) String Indexing and Slicing**

String indexing allows you to access individual characters in a string. You can do this by using square brackets and the location, or index, of the character you want to access. It's important to remember that Python starts indexes at 0. So to access the first character in a string, you would use the index [0]. If you try to access an index that’s larger than the length of your string, you’ll get an **IndexError**. This is because you’re trying to access something that doesn't exist! You can also access indexes from the end of the string going towards the start of the string by using negative values. The index [-1] would access the last character of the string, and the index [-2] would access the second-to-last character.

You can also access a portion of a string, called a slice or a substring. This allows you to access multiple characters of a string. You can do this by creating a range, using a colon as a separator between the start and end of the range, like [2:5]. This range is similar to the range() function we saw previously. It includes the first number, but goes to one less than the last number. You can also easily reference the start or end of the string by leaving one value blank. For example [:5] would give us all characters from the start to the fourth character in the string. We can also use [5:] to get the characters from the fourth character to the end of the string.

**(b) Basic String Methods**

In Python, strings are immutable. This means that they can't be modified. So if we wanted to fix a typo in a string, we can't simply modify the wrong character. We would have to create a new string with the typo corrected. We can also assign a new value to the variable holding our string.

If we aren't sure what the index of our typo is, we can use the string method *index* to locate it and return the index. Let's imagine we have the string **"lions tigers and bears"** in the variable **animals**. We can locate the index that contains the letter **g** using *animals.index("g")*, which will return the index; in this case 8. We can also use substrings to locate the index where the substring begins. *animals.index("bears")* would return 17, since that’s the start of the substring. If there’s more than one match for a substring, the index method will return the first match. If we try to locate a substring that doesn't exist in the string, we’ll receive a **ValueError** explaining that the substring was not found.

We can avoid a ValueError by first checking if the substring exists in the string. This can be done using the ***in*** keyword. We saw this keyword earlier when we covered *for* loops. In this case, it's a conditional that will be either True or False. If the substring is found in the string, it will be True. If the substring is not found in the string, it will be False. Using our previous variable **animals**, we can do **"horses"** **in animals** to check if the substring "horses" is found in our variable. In this case, it would evaluate to False, since horses aren’t included in our example string. If we did **"tigers" in animals**, we'd get True, since this substring is contained in our string.

**(c) Advanced String Methods**

We've covered a bunch of String class methods already, so let's keep building on those and run down some more advanced methods.

The string method **lower** will return the string with all characters changed to lowercase. The inverse of this is the **upper** method, which will return the string all in uppercase. Just like with previous methods, we call these on a string using dot notation, like **"this is a string".upper()**. This would return the string **"THIS IS A STRING"**. This can be super handy when checking user input, since someone might type in all lowercase, all uppercase, or even a mixture of cases.

You can use the **strip** method to remove surrounding whitespace from a string. Whitespace includes spaces, tabs, and newline characters. You can also use the methods  **lstrip** and **rstrip** to remove whitespace only from the left or the right side of the string, respectively.

The method **count** can be used to return the number of times a substring appears in a string. This can be handy for finding out how many characters appear in a string, or counting the number of times a certain word appears in a sentence or paragraph.

If you wanted to check if a string ends with a given substring, you can use the method **endswith**. This will return True if the substring is found at the end of the string, and False if not.

The **isnumeric** method can check if a string is composed of only numbers. If the string contains only numbers, this method will return True. We can use this to check if a string contains numbers before passing the string to the **int()** function to convert it to an integer, avoiding an error. Useful!

We took a look at string concatenation using the plus sign, earlier. We can also use the **join** method to concatenate strings. This method is called on a string that will be used to join a list of strings. The method takes a list of strings to be joined as a parameter, and returns a new string composed of each of the strings from our list joined using the initial string. For example, **" ".join(["This","is","a","sentence"])** would return the string **"This is a sentence"**.

The inverse of the join method is the **split** method. This allows us to split a string into a list of strings. By default, it splits by any whitespace characters. You can also split by any other characters by passing a parameter.

**(d) String Formatting**

You can use the **format** method on strings to concatenate and format strings in all kinds of powerful ways. To do this, create a string containing curly brackets, **{}**, as a placeholder, to be replaced. Then call the format method on the string using *.format()* and pass variables as parameters. The variables passed to the method will then be used to replace the curly bracket placeholders. This method automatically handles any conversion between data types for us.

If the curly brackets are empty, they’ll be populated with the variables passed in the order in which they're passed. However, you can put certain expressions inside the curly brackets to do even more powerful string formatting operations. You can put the name of a variable into the curly brackets, then use the names in the parameters. This allows for more easily readable code, and for more flexibility with the order of variables.

You can also put a formatting expression inside the curly brackets, which lets you alter the way the string is formatted. For example, the formatting expression **{:.2f}** means that you’d format this as a float number, with two digits after the decimal dot. The colon acts as a separator from the field name, if you had specified one. You can also specify text alignment using the greater than operator: **>**. For example, the expression **{:>3.2f}** would align the text three spaces to the right, as well as specify a float number with two decimal places. String formatting can be very handy for outputting easy-to-read textual output.

**(e) String Reference Cheat Sheet**

In Python, there are a lot of things you can do with strings. In this cheat sheet, you’ll find the most common string operations and string methods.

**String operations**

* len(string) Returns the length of the string
* for character in string Iterates over each character in the string
* if substring in string Checks whether the substring is part of the string
* string[i] Accesses the character at index i of the string, starting at zero
* string[i:j] Accesses the substring starting at index i, ending at index j-1. If i is omitted, it's 0 by default. If j is omitted, it's len(string) by default.

**String methods**

* string.lower() / string.upper() Returns a copy of the string with all lower / upper case characters
* string.lstrip() / string.rstrip() / string.strip() Returns a copy of the string without left / right / left or right whitespace
* string.count(substring) Returns the number of times substring is present in the string
* string.isnumeric() Returns True if there are only numeric characters in the string. If not, returns False.
* string.isalpha() Returns True if there are only alphabetic characters in the string. If not, returns False.
* string.split() / string.split(delimiter) Returns a list of substrings that were separated by whitespace / delimiter
* string.replace(old, new) Returns a new string where all occurrences of old have been replaced by new.
* delimiter.join(list of strings) Returns a new string with all the strings joined by the delimiter

Check out the official documentation for [all available String methods](https://docs.python.org/3/library/stdtypes.html#string-methods).

**(f) Formatting Strings Cheat Sheet**

Python offers different ways to format strings. In the video, we explained the format() method. In this reading, we'll highlight three different ways of formatting strings. For this course you only need to know the format() method. But on the internet, you might find any of the three, so it's a good idea to know that the others exist.

**Using the format() method**

The format method returns a copy of the string where the {} placeholders have been replaced with the values of the variables. These variables are converted to strings if they weren't strings already. Empty placeholders are replaced by the variables passed to format in the same order.

# "base string with {} placeholders".format(variables)

example = "format() method"

formatted\_string = "this is an example of using the {} on a string".format(example)

print(formatted\_string)

"""Outputs:

this is an example of using the format() method on a string

"""

If the placeholders indicate a number, they’re replaced by the variable corresponding to that order (starting at zero).

# "{0} {1}".format(first, second)

first = "apple"

second = "banana"

third = "carrot"

formatted\_string = "{0} {2} {1}".format(first, second, third)

print(formatted\_string)

"""Outputs:

apple carrot banana

"""

If the placeholders indicate a field name, they’re replaced by the variable corresponding to that field name. This means that parameters to format need to be passed indicating the field name.

# "{var1} {var2}".format(var1=value1, var2=value2)

"{:exp1} {:exp2}".format(value1, value2)

If the placeholders include a colon, what comes after the colon is a formatting expression. See below for the expression reference.

Official documentation for [the format string syntax](https://docs.python.org/3/library/string.html#formatstrings)

# {:d} integer value

'{:d}'.format(10.5) → '10'

**Formatting expressions**

| **Expr** | **Meaning** | | **Example** |
| --- | --- | --- | --- |
| {:d} | integer value | '{:d}'.format(10.5) → '10' | |
| {:.2f} | floating point with that many decimals | '{:.2f}'.format(0.5) → '0.50' | |
| {:.2s} | string with that many characters | '{:.2s}'.format('Python') → 'Py' | |
| {:<6s} | string aligned to the left that many spaces | '{:<6s}'.format('Py') → 'Py    ' | |
| {:>6s} | string aligned to the right that many spaces | '{:>6s}'.format('Py') → '    Py' | |
| {:^6s} | string centered in that many spaces | '{:^6s}'.format('Py') → '  Py ' | |

Check out the official documentation for [all available expressions](https://docs.python.org/3/library/string.html#format-specification-mini-language).

**Old string formatting (Optional)**

The format() method was introduced in Python 2.6. Before that, the % (modulo) operator could be used to get a similar result. While this method is **no longer recommended** for new code, you might come across it in someone else's code. This is what it looks like:

"base string with %s placeholder" % variable

The % (modulo) operator returns a copy of the string where the placeholders indicated by %  followed by a formatting expression are replaced by the variables after the operator.

"base string with %d and %d placeholders" % (value1, value2)

To replace more than one value, the values need to be written between parentheses. The formatting expression needs to match the value type.

"%(var1) %(var2)" % {var1:value1, var2:value2}

Variables can be replaced by name using a dictionary syntax (we’ll learn about dictionaries in an upcoming video).

"Item: %s - Amount: %d - Price: %.2f" % (item, amount, price)

The formatting expressions are mostly the same as those of the format() method.

Check out the official documentation for [old string formatting](https://docs.python.org/3/library/stdtypes.html#old-string-formatting).

**Formatted string literals (Optional)**

This feature was added in Python 3.6 and isn’t used a lot yet. Again, it's included here in case you run into it in the future, but it's not needed for this or any upcoming courses.

A formatted string literal or f-string is a string that starts with 'f' or 'F' before the quotes. These strings might contain {} placeholders using expressions like the ones used for format method strings.

The important difference with the format method is that it takes the value of the variables from the current context, instead of taking the values from parameters.

Examples:

>>> name = "Micah"

>>> print(f'Hello {name}')

Hello Micah

>>> item = "Purple Cup"

>>> amount = 5

>>> price = amount \* 3.25

>>> print(f'Item: {item} - Amount: {amount} - Price: {price:.2f}')

Item: Purple Cup - Amount: 5 - Price: 16.25

Check out the official documentation for [f-strings](https://docs.python.org/3/reference/lexical_analysis.html#f-strings).

1. Lists

**(a) Lists Defined**

Lists in Python are defined using square brackets, with the elements stored in the list separated by commas: **list = ["This", "is", "a", "list"]**. You can use the **len()** function to return the number of elements in a list: **len(list)** would return **4**. You can also use the **in** keyword to check if a list contains a certain element. If the element is present, it will return a True boolean. If the element is not found in the list, it will return False. For example, **"This" in list** would return True in our example. Similar to strings, lists can also use indexing to access specific elements in a list based on their position. You can access the first element in a list by doing **list[0]**, which would allow you to access the string **"This"**.

In Python, lists and strings are quite similar. They’re both examples of sequences of data. Sequences have similar properties, like (1) being able to iterate over them using **for loops**; (2) support indexing; (3) using the **len** function to find the length of the sequence; (4) using the plus operator **+** in order to concatenate; and (5) using the **in** keyword to check if the sequence contains a value. Understanding these concepts allows you to apply them to other sequence types as well.

**(b) Modifying Lists**

While lists and strings are both sequences, a big difference between them is that lists are mutable. This means that the contents of the list can be changed, unlike strings, which are immutable. You can add, remove, or modify elements in a list.

You can add elements to the end of a list using the **append** method. You call this method on a list using dot notation, and pass in the element to be added as a parameter. For example, **list.append("New data")** would add the string "New data" to the end of the list called list.

If you want to add an element to a list in a specific position, you can use the method **insert**. The method takes two parameters: the first specifies the index in the list, and the second is the element to be added to the list. So **list.insert(0, "New data")** would add the string "New data" to the front of the list. This wouldn't overwrite the existing element at the start of the list. It would just shift all the other elements by one. If you specify an index that’s larger than the length of the list, the element will simply be added to the end of the list.

You can remove elements from the list using the **remove** method. This method takes an element as a parameter, and removes the first occurrence of the element. If the element isn’t found in the list, you’ll get a **ValueError** error explaining that the element was not found in the list.

You can also remove elements from a list using the **pop** method. This method differs from the remove method in that it takes an index as a parameter, and returns the element that was removed. This can be useful if you don't know what the value is, but you know where it’s located. This can also be useful when you need to access the data and also want to remove it from the list.

Finally, you can change an element in a list by using indexing to overwrite the value stored at the specified index. For example, you can enter **list[0] = "Old data"** to overwrite the first element in a list with the new string "Old data".

**(c) Tuples**

As we mentioned earlier, strings and lists are both examples of sequences. Strings are sequences of characters, and are immutable. Lists are sequences of elements of any data type, and are mutable. The third sequence type is the tuple. Tuples are like lists, since they can contain elements of any data type. But unlike lists, tuples are immutable. They’re specified using parentheses instead of square brackets.

You might be wondering why tuples are a thing, given how similar they are to lists. Tuples can be useful when we need to ensure that an element is in a certain position and will not change. Since lists are mutable, the order of the elements can be changed on us. Since the order of the elements in a tuple can't be changed, the position of the element in a tuple can have meaning. A good example of this is when a function returns multiple values. In this case, what gets returned is a tuple, with the return values as elements in the tuple. The order of the returned values is important, and a tuple ensures that the order isn’t going to change. Storing the elements of a tuple in separate variables is called unpacking. This allows you to take multiple returned values from a function and store each value in its own variable.

**(d) Iterating Over Lists Using Enumerate**

When we covered *for* loops, we showed the example of iterating over a list. This lets you iterate over each element in the list, exposing the element to the for loop as a variable. But what if you want to access the elements in a list, along with the index of the element in question? You can do this using the **enumerate()** function. The enumerate() function takes a list as a parameter and returns a tuple for each element in the list. The first value of the tuple is the index and the second value is the element itself.

**(e) List Comprehensions**

You can create lists from sequences using a for loop, but there’s a more streamlined way to do this: **list comprehension**. List comprehensions allow you to create a new list from a sequence or a range in a single line.

For example, **[ x\*2 for x in range(1,11) ]** is a simple list comprehension. This would iterate over the range 1 to 10, and multiply each element in the range by 2. This would result in a list of the multiples of 2, from 2 to 20.

You can also use conditionals with list comprehensions to build even more complex and powerful statements. You can do this by appending an if statement to the end of the comprehension. For example, **[ x for x in range(1,101) if x % 10 == 0 ]** would generate a list containing all the integers divisible by 10 from 1 to 100. The if statement we added here evaluates each value in the range from 1 to 100 to check if it’s evenly divisible by 10. If it is, it gets added to the list.

List comprehensions can be really powerful, but they can also be super complex, resulting in code that’s hard to read. Be careful when using them, since it might make it more difficult for someone else looking at your code to easily understand what the code is doing.

**(f) Lists and Tuples Operations Cheat Sheet**

Lists and tuples are both sequences, so they share a number of sequence operations. But, because lists are mutable, there are also a number of methods specific just to lists. This cheat sheet gives you a run down of the common operations first, and the list-specific operations second.

**Common sequence operations**

* len(sequence) Returns the length of the sequence
* for element in sequence Iterates over each element in the sequence
* if element in sequence Checks whether the element is part of the sequence
* sequence[i] Accesses the element at index i of the sequence, starting at zero
* sequence[i:j] Accesses a slice starting at index i, ending at index j-1. If i is omitted, it's 0 by default. If j is omitted, it's len(sequence) by default.
* for index, element in enumerate(sequence) Iterates over both the indexes and the elements in the sequence at the same time

Check out the [official documentation for sequence operations](https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range).

**List-specific operations and methods**

* list[i] = x Replaces the element at index i with x
* list.append(x) Inserts x at the end of the list
* list.insert(i, x) Inserts x at index i
* list.pop(i) Returns the element a index i, also removing it from the list. If i is omitted, the last element is returned and removed.
* list.remove(x) Removes the first occurrence of x in the list
* list.sort() Sorts the items in the list
* list.reverse() Reverses the order of items of the list
* list.clear() Removes all the items of the list
* list.copy() Creates a copy of the list
* list.extend(other\_list) Appends all the elements of other\_list at the end of list

Most of these methods come from the fact that lists are mutable sequences. For more info, see the [official documentation for mutable sequences](https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types) and the [list specific documentation](https://docs.python.org/3/library/stdtypes.html#lists).

**List comprehension**

* [expression for variable in sequence] Creates a new list based on the given sequence. Each element is the result of the given expression.
* [expression for variable in sequence if condition] Creates a new list based on the given sequence. Each element is the result of the given expression; elements only get added if the condition is true.

1. Dictionaries Defined

Dictionaries are another data structure in Python. They’re similar to a list in that they can be used to organize data into collections. However, data in a dictionary isn't accessed based on its position. Data in a dictionary is organized into pairs of keys and values. You use the key to access the corresponding value. Where a list index is always a number, a dictionary key can be a different data type, like a string, integer, float, or even tuples.

When creating a dictionary, you use curly brackets: **{}**. When storing values in a dictionary, the key is specified first, followed by the corresponding value, separated by a colon. For example, **animals = {"bears":10, "lions":1, "tigers":2}** creates a dictionary with three key value pairs, stored in the variable animals. The key "bears" points to the integer value 10, while the key "lions" points to the integer value 1, and "tigers" points to the integer 2. You can access the values by referencing the key, like this: **animals["bears"]**. This would return the integer 10, since that’s the corresponding value for this key.

You can also check if a key is contained in a dictionary using the **in** keyword. Just like other uses of this keyword, it will return True if the key is found in the dictionary; otherwise it will return False.

Dictionaries are mutable, meaning they can be modified by adding, removing, and replacing elements in a dictionary, similar to lists. You can add a new key value pair to a dictionary by assigning a value to the key, like this: **animals["zebras"] = 2**. This creates the new key in the animal dictionary called zebras, and stores the value 2. You can modify the value of an existing key by doing the same thing. So **animals["bears"] = 11** would change the value stored in the bears key from 10 to 11. Lastly, you can remove elements from a dictionary by using the **del** keyword. By doing **del animals["lions"]** you would remove the key value pair from the animals dictionary.

**(a) Iterating Over Dictionaries**

You can iterate over dictionaries using a *for* loop, just like with strings, lists, and tuples. This will iterate over the sequence of keys in the dictionary. If you want to access the corresponding values associated with the keys, you could use the keys as indexes. Or you can use the **items** method on the dictionary, like **dictionary.items()**. This method returns a tuple for each element in the dictionary, where the first element in the tuple is the key and the second is the value.

If you only wanted to access the keys in a dictionary, you could use the **keys()** method on the dictionary: **dictionary.keys()**. If you only wanted the values, you could use the **values()** method: **dictionary.values()**.

**(b) Dictionary Methods Cheat Sheet**

**Definition**

x = {key1:value1, key2:value2}

**Operations**

* len(dictionary) - Returns the number of items in the dictionary
* for key in dictionary - Iterates over each key in the dictionary
* for key, value in dictionary.items() - Iterates over each key,value pair in the dictionary
* if key in dictionary - Checks whether the key is in the dictionary
* dictionary[key] - Accesses the item with key key of the dictionary
* dictionary[key] = value - Sets the value associated with key
* del dictionary[key] - Removes the item with key key from the dictionary

**Methods**

* dict.get(key, default) - Returns the element corresponding to key, or default if it's not present
* dict.keys() - Returns a sequence containing the keys in the dictionary
* dict.values() - Returns a sequence containing the values in the dictionary
* dict.update(other\_dictionary) - Updates the dictionary with the items coming from the other dictionary. Existing entries will be replaced; new entries will be added.
* dict.clear() - Removes all the items of the dictionary

Check out the [official documentation for dictionary operations and methods](https://docs.python.org/3/library/stdtypes.html#mapping-types-dict). s

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

OBJECT ORIENTED PROGRAMMING

**(a) Object-Oriented Programming Defined**

In object-oriented programming, concepts are modeled as classes and objects. An idea is defined using a class, and an instance of this class is called an object. Almost everything in Python is an object, including strings, lists, dictionaries, and numbers. When we create a list in Python, we’re creating an object which is an instance of the list class, which represents the concept of a list. Classes also have attributes and methods associated with them. Attributes are the characteristics of the class, while methods are functions that are part of the class.

**(b) Classes and Objects in Detail**

We can use the **type()** function to figure out what class a variable or value belongs to. For example, **type(" ")** tells us that this is a string class. The only attribute in this case is the string value, but there are a bunch of methods associated with the class. We've seen the **upper()** method, which returns the string in all uppercase, as well as **isnumeric()** which returns a boolean telling us whether or not the string is a number. You can use the **dir()** function to print all the attributes and methods of an object. Each string is an instance of the string class, having the same methods of the parent class. Since the content of the string is different, the methods will return different values. You can also use the **help()** function on an object, which will return the documentation for the corresponding class. This will show all the methods for the class, along with parameters the methods receive, types of return values, and a description of the methods.

**(c) Defining Classes (Optional)**

We can create and define our classes in Python similar to how we define functions. We start with the class keyword, followed by the name of our class and a colon. Python style guidelines recommend class names to start with a capital letter. After the class definition line is the class body, indented to the right. Inside the class body, we can define attributes for the class.

Let's take our Apple class example:

>>> class Apple:

... color = ""

... flavor = ""

We can create a new instance of our new class by assigning it to a variable. This is done by calling the class name as if it were a function. We can set the attributes of our class instance by accessing them using dot notation. Dot notation can be used to set or retrieve object attributes, as well as call methods associated with the class.

>>> jonagold = Apple()

>>> jonagold.color = "red"

>>> jonagold.flavor = "sweet"

We created an Apple instance called jonagold, and set the color and flavor attributes for this Apple object. We can create another instance of an Apple and set different attributes to differentiate between two different varieties of apples.

>>> golden = Apple()

>>> golden.color = "Yellow"

>>> golden.flavor = "Soft"

We now have another Apple object called golden that also has color and flavor attributes. But these attributes have different values.

**(d) What Is a Method?**

Calling methods on objects executes functions that operate on attributes of a specific instance of the class. This means that calling a method on a list, for example, only modifies that instance of a list, and not all lists globally. We can define methods within a class by creating functions inside the class definition. These instance methods can take a parameter called **self** which represents the instance the method is being executed on. This will allow you to access attributes of the instance using dot notation, like **self.name**, which will access the name attribute of that specific instance of the class object. When you have variables that contain different values for different instances, these are called instance variables.

**(e) Special Methods**

Instead of creating classes with empty or default values, we can set these values when we create the instance. This ensures that we don't miss an important value and avoids a lot of unnecessary lines of code. To do this, we use a special method called a **constructor**. Below is an example of an Apple class with a constructor method defined.

>>> class Apple:

... def \_\_init\_\_(self, color, flavor):

... self.color = color

... self.flavor = flavor

When you call the name of a class, the constructor of that class is called. This constructor method is always named **\_\_init\_\_**. You might remember that special methods start and end with two underscore characters. In our example above, the constructor method takes the self variable, which represents the instance, as well as color and flavor parameters. These parameters are then used by the constructor method to set the values for the current instance. So we can now create a new instance of the Apple class and set the color and flavor values all in go:

>>> jonagold = Apple("red", "sweet")

>>> print(jonagold.color)

Red

In addition to the **\_\_init\_\_** constructor special method, there is also the **\_\_str\_\_** special method. This method allows us to define how an instance of an object will be printed when it’s passed to the print() function. If an object doesn’t have this special method defined, it will wind up using the default representation, which will print the position of the object in memory. Not super useful. Here is our Apple class, with the **\_\_str\_\_** method added:

>>> class Apple:

... def \_\_init\_\_(self, color, flavor):

... self.color = color

... self.flavor = flavor

... def \_\_str\_\_(self):

... return "This apple is {} and its flavor is {}".format(self.color, self.flavor)

Now, when we pass an Apple object to the print function, we get a nice formatted string:

>>> jonagold = Apple("red", "sweet")

>>> print(jonagold)

This apple is red and its flavor is sweet

This apple is red and its flavor is sweet

It's good practice to think about how your class might be used and to define a \_\_str\_\_ method when creating objects that you may want to print later.

**(f) Documenting with Docstrings**

The Python **help** function can be super helpful for easily pulling up documentation for classes and methods. We can call the **help** function on one of our classes, which will return some basic info about the methods defined in our class:

>>> class Apple:

... def \_\_init\_\_(self, color, flavor):

... self.color = color

... self.flavor = flavor

... def \_\_str\_\_(self):

... return "This apple is {} and its flavor is {}".format(self.color, self.flavor)

...

>>> help(Apple)

Help on class Apple in module \_\_main\_\_:

class Apple(builtins.object)

| Methods defined here:

|

| \_\_init\_\_(self, color, flavor)

| Initialize self. See help(type(self)) for accurate signature.

|

| \_\_str\_\_(self)

| Return str(self).

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

We can add documentation to our own classes, methods, and functions using **docstrings**. A docstring is a short text explanation of what something does. You can add a docstring to a method, function, or class by first defining it, then adding a description inside triple quotes. Let's take the example of this function:

>>> def to\_seconds(hours, minutes, seconds):

... """Returns the amount of seconds in the given hours, minutes and seconds."""

... return hours\*3600+minutes\*60+seconds

We have our function called to\_seconds on the first line, followed by the docstring which is indented to the right and wrapped in triple quotes. Last up is the function body. Now, when we call the help function on our to\_seconds function, we get a handy description of what the function does:

>>> help(to\_seconds)

Help on function to\_seconds in module \_\_main\_\_:

to\_seconds(hours, minutes, seconds)

Returns the amount of seconds in the given hours, minutes and seconds.

Docstrings are super useful for documenting our custom classes, methods, and functions, but also when working with new libraries or functions. You'll be extremely grateful for docstrings when you have to work with code that someone else wrote!

**(g) Object Inheritance**

In object-oriented programming, the concept of inheritance allows you to build relationships between objects, grouping together similar concepts and reducing code duplication. Let's create a custom Fruit class with color and flavor attributes:

>>> class Fruit:

...     def \_\_init\_\_(self, color, flavor):

...         self.color = color

...         self.flavor = flavor

We defined a Fruit class with a constructor for color and flavor attributes. Next, we'll define an Apple class along with a new Grape class, both of which we want to inherit properties and behaviors from the Fruit class:

>>> class Apple(Fruit):

...     pass

>>> class Grape(Fruit):

...     pass

In Python, we use parentheses in the class declaration to have the class inherit from the Fruit class. So in this example, we’re instructing our computer that both the Apple class and Grape class inherit from the Fruit class. This means that they both have the same constructor method which sets the color and flavor attributes. We can now create instances of our Apple and Grape classes:

>>> carnelian = Grape("purple", "sweet")

>>> print(granny\_smith.flavor)

tart

>>> print(carnelian.color)

Purple

Inheritance allows us to define attributes or methods that are shared by all types of fruit without having to define them in each fruit class individually. We can then also define specific attributes or methods that are only relevant for a specific type of fruit. Let's look at another example, this time with animals:

>>> class Animal:

...     sound = ""

...     def \_\_init\_\_(self, name):

...         self.name = name

...     def speak(self):

...         print("{sound} I'm {name}! {sound}".format(

...             name=self.name, sound=self.sound))

...

>>> class Piglet(Animal):

...     sound = "Oink!"

>>> class Cow(Animal):

...     sound = "Moooo"

We defined a parent class, Animal, with two animal types inheriting from that class: Piglet and Cow. The parent Animal class has an attribute to store the sound the animal makes, and the constructor class takes the name that will be assigned to the instance when it's created. There is also the speak method, which will print the name of the animal along with the sound it makes. We defined the Piglet and Cow classes, which inherit from the Animal class, and we set the sound attributes for each animal type. Now, we can create instances of our Piglet and Cow classes and have them speak:

>>> hamlet = Piglet("Hamlet")

>>> hamlet.speak()

Oink! I'm Hamlet! Oink!

...

>>> class Cow(Animal):

...     sound = "Moooo"

...

>>> milky = Cow("Milky White")

>>> milky.speak()

Moooo I'm Milky White! Moooo

We create instances of both the Piglet and Cow class, and set the names for our instances. Then we call the speak method of each instance, which results in the formatted string being printed; it includes the sound the animal type makes, along with the instance name we assigned.

**(h) Object Composition**

You can have a situation where two different classes are related, but there is no inheritance going on. This is referred to as **composition** -- where one class makes use of code contained in another class. For example, imagine we have a **Package** class which represents a software package. It contains attributes about the software package, like name, version, and size. We also have a **Repository** class which represents all the packages available for installation. While there’s no inheritance relationship between the two classes, they are related. The Repository class will contain a dictionary or list of Packages that are contained in the repository. Let's take a look at an example Repository class definition:

>>> class Repository:

...      def \_\_init\_\_(self):

...          self.packages = {}

...      def add\_package(self, package):

...          self.packages[package.name] = package

...      def total\_size(self):

...          result = 0

...          for package in self.packages.values():

...              result += package.size

...          return result

In the constructor method, we initialize the packages dictionary, which will contain the package objects available in this repository instance. We initialize the dictionary in the constructor to ensure that every instance of the Repository class has its own dictionary.

We then define the add\_package method, which takes a Package object as a parameter, and then adds it to our dictionary, using the package name attribute as the key.

Finally, we define a total\_size method which computes the total size of all packages contained in our repository. This method iterates through the values in our repository dictionary and adds together the size attributes from each package object contained in the dictionary, returning the total at the end. In this example, we’re making use of Package attributes within our Repository class. We’re also calling the values() method on our packages dictionary instance. Composition allows us to use objects as attributes, as well as access all their attributes and methods.

**(i) Augmenting Python with Modules**

Python modules are separate files that contain classes, functions, and other data that allow us to import and make use of these methods and classes in our own code. Python comes with a lot of modules out of the box. These modules are referred to as the Python Standard Library. You can make use of these modules by using the **import** keyword, followed by the module name. For example, we'll import the **random** module, and then call the **randint** function within this module:

>>> import random

>>> random.randint(1,10)

8

>>> random.randint(1,10)

7

>>> random.randint(1,10)

1

This function takes two integer parameters and returns a random integer between the values we pass it; in this case, 1 and 10. You might notice that calling functions in a module is very similar to calling methods in a class. We use dot notation here too, with a period between the module and function names.

Let's take a look at another module: **datetime**. This module is super helpful when working with dates and times.

>>> import datetime

>>> now = datetime.datetime.now()

>>> type(now)

<class 'datetime.datetime'>

>>> print(now)

2019-04-24 16:54:55.155199

First, we import the module. Next, we call the **now()** method which belongs to the **datetime** class contained within the **datetime** module. This method generates an instance of the datetime class for the current date and time. This instance has some methods which we can call:

2019-04-24 16:54:55.155199

>>> now.year

2019

>>> print(now + datetime.timedelta(days=28))

2019-05-22 16:54:55.155199

When we call the print function with an instance of the datetime class, we get the date and time printed in a specific format. This is because the datetime class has a **\_\_str\_\_** method defined which generates the formatted string we see here. We can also directly call attributes and methods of the class, as with **now.year** which returns the year attribute of the instance.

Lastly, we can access other classes contained in the datetime module, like the **timedelta** class. In this example, we’re creating an instance of the timedelta class with the parameter of 28 days. We’re then adding this object to our instance of the datetime class from earlier and printing the result. This has the effect of adding 28 days to our original datetime object.

**(j) Supplemental Reading for Code Reuse**

The official Python documentation lists all the modules included in the standard library. It even has a turtle in it...

[Pypi](https://pypi.org/) is the Python repository and index of an impressive number of modules developed by Python programmers around the world.