Cheat Sheet

CODE University of Applied Sciences

Beginner Python Course

How to use this document

Each concept is given in the order it appears within the lessons. You can review the concepts for each lesson you have completed to check your knowledge, or use the search function or contents table to navigate to the relevant entry (for example, if you need a reminder of something from a previous lesson when you are further on in the course).

Contents

Lesson 1

The print() function

- Displays text on screen; e.g:
  `print("Hello, World!")`

Errors

- Tell you that there is something wrong with the code you are trying to run.
- The line number tells you where in the code the error is and the ^ symbol indicates where the problem is in that line.

Comments

- Use the `#` symbol to write a comment in code that will be ignored by Python, but that can help make your code understandable; e.g.:
  `#This is a comment`

Syntax

- The rules that govern the use of words, punctuation, and spaces in that language.
- E.g. code blocks in Python must be indented and function names are always followed by parentheses.

Strings (data type)

- Any characters (letters, numbers, or punctuation, etc.) between single or double quotes; e.g.
  `"Hello, World!"` or `'Hello, World!'`

Lesson 2

Integers (data type)

- Integers are whole numbers. E.g. 42

Floats (data type)

- Floats are numbers with decimal points ("floating point numbers"). E.g. 42.33

The int() function

- Converts strings or floats into integers. E.g. `int("42")` or `int(42.33)` both output 42.

The float() function

- Converts strings or integers into floating point numbers.
- E.g. `float("42")` or `float(42)` or `float(41 +1)` all output the float 42.0.

The str() function

- Converts various data types into strings. E.g. `str(42)` outputs `"42"`.

Variables

- Used to store data (whether number, string, or other data type) for reuse later.
- Assigned with a single equal sign; e.g.: `variable = 42`
- A variable name can be anything that is not a reserved word.
- The value of a variable can be changed.

Reserved words

- Certain words that have a predefined meaning within Python, like “print” in the `print()` function, and that therefore cannot be used as variable names.

The input() function

- Allows the user to input data into the program.
- Adding a string between the parenthesis will display it before the input cell on screen; e.g.:
  `input("Please enter a number:")`
- Can be assigned to a variable to use the user input later; e.g.:
  `user_input = input("Please enter a number:")`
- Will always output the user input as a string (even if they input a number). This means you may need to convert that input back into a number to work with it; e.g.:
  `int(input("Please enter a number:"))`

Adding strings

- You can use the + sign to add strings together (including those stored in variables); e.g.:
  `print("Hello, " + user_name)`

Lesson 3

Booleans (data type)

- A data type with two possible values: `True` or `False`.
- Must start with a capital letter.
- 0 or an empty string are considered “falsy”.
- 1 or any higher number, or a string with at least one character is “truthy”.

Comparison operators (>, <, ==, >=, <=, !=)

- Used to compare two pieces of data.
- E.g. whether a value is:
  `>` (greater than)
  `<` (less than)
  `==` (equal to)
  `>=` (greater than or equal to)
  `<=` (less than or equal to)
  `!=` (not equal to; this is called the bang operator)
- Can be used to compare numeric values or (in the case of `==` and `!=`) strings.

Logical operators (and, or, not)

- Used to combine or modify boolean values.
- Used in conditions, boolean expressions, and loops to check whether a `and` b are True, whether a `or` b is True or whether something is `not` True; e.g.:
  The following outputs True because both conditions are True:
  ```
  x = 5
  y = 10
  result = (x > 0) and (y < 15)
  ```
  The following outputs True because one of the two conditions is True:
  ```
  x = 5
  y = 10
  result = (x > 0) or (y < 5)
  ```
  The following outputs True because the condition is False:
  ```
  x = 5
  result = not (x < 0)
  ```

Membership operator (in)

- Used to check whether a certain value appears in a piece of data (e.g. a string or list); e.g:
  ```
  "World" in "Hello, World!"
  1 in [1, 2, 3]
  ```

Modulo operator (%)

- Used to return the remainder of a division; e.g.:
  `5 % 2` returns 1 (2 goes twice into 5 with a remainder of 1)
  `10 % 5` returns 0 (5 goes twice into 10 with a remainder of 0)

Lesson 4

Control flow

- The order in which code is executed.
- Managed using conditional statements and loops.

Conditional statements (if, elif, else)

- Lets you write code that only executes if a certain condition is met; e.g. “if a is equal to b, then do the following”.
- Formatting is important: there must be a colon at the end of each `if/elif/else` line, and the code below must be indented; e.g.:
```
if first_variable > second_variable:
     print("The first value is greater.")
elif first_variable < second_variable:
     print("The second value is greater.")
else:
     print("The values are equal.")
```
- Remember that a non-empty string or any number above 0 is “truthy”, so you can use an if statement to check whether a variable has a “truthy” value; e.g.:
```
if user_name:
     print("Hello, " + user_name)
else:
     print("You need to enter a name.")
```

Code blocks

- Defined by colons and indentation (see “Conditions” above).
- Tells Python what is and is not part of e.g. a condition (`if`), loop (`while` or `for`), or function definition (`def`).

The .isnumeric() method

- Checks whether or not the value stored in string is entirely numeric (returns `True` if all characters are numeric or `False` if none or only some are); e.g. the first line below returns `True` and the second returns `False`:
```
"42".isnumeric()
"42 is the meaning of life".isnumeric()
```
- Can be used with a variable that contains a string; e.g.:
```
user_age.isnumeric()
```

Lesson 5

Functions

- Block of reusable code that performs a specific task.
- There are inbuilt functions—like `print()` and `input()`—and you can also define your own.
- Can be named anything that's not a reserved word and called by its name followed by parentheses.
- New functions are defined using the keyword `def`, a colon, and indented code block; e.g.:

```
def print_greeting():
      name = input("Name:")
      print("Hello, " + name)
```

Function arguments

- Variable defined within a function's parentheses (also known as parameters).
- Each time the function is called, the argument can be changed; e.g:

```
def add_two_numbers(number1, number2):
      return number1 + number2
result = add_two_numbers(4, 2)
print(result)
```

Methods

- Type of function that works with specific objects or data types (e.g. numbers, strings, or lists).
- Followed by parentheses and called by appending it to the object name with dot notation; e.g. the following checks whether the value stored in `variable_name` is numeric and returns `True` or `False`: `variable_name.isnumeric()`

Block scope

- Refers to the accessibility of variables or functions. E.g. if a variable is assigned a value within the local scope of a function (the code block in which a function is defined), it will only be available in the parent or global scope (the outer or main code blocks) if the function returns that value.

Return statement

- Returns a value generated or assigned within a function to be used in the parent scope.
- Using `return` ends the function execution; it should therefore be on the final line of a function's code block, as any code below will be ignored.

Refactoring

- Rewriting "messy" or inefficient code into something "cleaner"; e.g. by using functions.

The random.randint() method

- Generates a random integer within a specified range.
- You must first `import random` to access this feature.
- Range is specified in the function arguments; e.g. the following will return a random number between 1 and 10: `random.randint(1, 10)`

Lesson 6

While-loops

- Repeats the code inside the while-loop code block for as long as (`while`) a condition is `True`.
- You can make an infinite loop with `while True:` (this will only stop running if you use the `break` command).
- The while-loop code block is signaled by a colon and indentation; e.g.:

```
while a == b:
      print("The values are equal.")
```

The break command

- Stops a loop running; e.g:

```
while True:
      print("Hello, World!")
      break
```

- Using the `return` command within a function within a loop also breaks the loop.

The continue command

- Skips the code below and jumps to the top of the loop to start a new iteration; e.g.:

```
while True:
      user_name = input("Name:")
      if not user_name:
            Continue
      else:
            Break
```

The time.sleep() method

- Allows you to introduce a delay into the running of the program.
- You must first `import time` to use this.
- The delay is given in seconds in the argument of the function; e.g. the following would introduce a delay of 3 seconds: `time.sleep(3)`

The len() function

- Counts the length of an object; e.g. the number of characters in a string or the number of items in a list.
- Outputs an integer, so `len("Hello, World!")` outputs 13.

Using math and while-loops to create a counter

- You can use basic math to create a counter to count rounds.
- Assign a variable a numeric value and then add or subtract 1 for each round.
- The loop can be set to run until a certain value is reached; e.g.:

```
counter = 0
while counter < 6:
      print(counter)
      counter = counter + 1
```

Lesson 7

Lists (complex data type)

- Stores multiple pieces of data (whether numbers, strings, or other data types).
- Also known as arrays in other programming languages.
- Lists are assigned with square brackets, and list items are separated by commas; e.g. `my_list = [1, 2, 3, 4, 5]`
- You can create an empty list by just using square brackets; e.g. `my_list = []`
- It's good practice to put list items of longer lists on separate lines and indent for ease of readability.

Selecting list items

- You select a list item by calling the list name with the index number in square brackets; e.g. `my_list[3]`
- The index starts at 0, so `my_list[3]` outputs the number 4 from the following list: `my_list = [1, 2, 3, 4, 5]`

Changing list items

- You use the index of the list item to assign a new value; e.g. `my_list[0] = "Hello"`
- This only works with list items that already exist.

The .append() method (adding list items)

- Adds items to an existing list. E.g. the following would add the integer 42 to the end of `my_list`: `my_list.append(42)`

The .remove() method

- Removes the first instance of a list item via its exact value. E.g. the following removes the first instance of the number 42 from `my_list`: `my_list.remove(42)`

The .pop() method

- Removes a list item via its index number and returns it. E.g. the following will remove the first item of the list and return it: `my_list.pop(0)`

Error handling

The practice of writing programs that anticipate, detect, and manage errors or exceptions that may occur during execution. For example, if user input should be of a certain data type or have certain characteristics (for example, it should be numerical, or should be a string with at least a certain number of characters), the program should check for this and display a message to the user to correct the input if necessary. When writing a program it's a good idea to test all possible scenarios to make sure that it works as intended.

Using the membership operator (in) with an if-else statement

- Can be used to perform an action only if a certain value appears `in` a list (or string); e.g.:

```
pizza_ingredients = ["dough", "tomato sauce", "cheese"]
if "cheese" in pizza_ingredients:
    print("This pizza has dairy in it.")
else:
    print("There is no dairy in this pizza!")
```

Lesson 8

For-loops

- Iterates code for each item in a sequence.
- You create a loop variable (or iteration variable) to stand in for each item in a sequence; e.g. the following code would print the value of each item stored in `my_list` (but you could use any word in place of `item`):
  ```
  for item in my_list:
       print(item)
  ```

Tuples (complex data type)

- Like a list, a tuple stores multiple pieces of data (whether numbers, strings, or other data types), but it is *immutable*, meaning that the values cannot be changed (unlike lists, which can be changed).
- Tuples are assigned with normal brackets, and tuple items are separated by commas; e.g.:
  ```
  my_tuple = (1, 2, 3, 4, 5)
  ```

Unpacking / Destructuring

- Enables you to extract multiple items from a list, tuple, or dictionary and assign them separate variables all at once; e.g. the following assigns separate variables to each of the names in the list:
  ```
  name_list = ["Rowan", "Sam", "Mark"]
  first_name, second_name, third_name = list_of_names
  ```

The enumerate() function

- Returns a two-item tuple for each item in a sequence that includes the index number and the item value; e.g:
  ```
  my_list = ["apple", "banana", "cherry"]
  for index, value in enumerate(my_list):
       print(index, value)
  ```
- The above code outputs:
  ```
  0 apple
  1 banana
  2 cherry
  ```

Lesson 9

Dictionaries (complex data type)

- Pairs data into keys and values.
- The value in a dictionary can be any data type: strings, numbers, lists, tuples (or even other dictionaries) and can be changed.
- The key is used to access values; each key must be unique and cannot be changed. It can be any immutable data type (string, integer, etc.)
- Assigned using curly brackets; e.g. `my_dictionary = {}` creates an empty dictionary.
- Keys and values are separated by a colon; key-value pairs are separated by commas; e.g.:
  ```
  dictionary_with_strings_as_keys = {"key_1": 10, "key_2": 20}
  dictionary_with_ints_as_keys = {1: 10, 2: 20}
  ```
- As with lists, it's good practice to write dictionary items on separate lines and indent, for ease of readability.

Selecting dictionary items

- You use the key to access a value in a dictionary.
- This is done by calling the dictionary name and then putting the key name in square brackets; e.g. `dictionary_with_strings_as_keys["key_1"]` accesses whatever value is stored in `"key_1"` and `dictionary_with_ints_as_keys[1]` accesses whatever value is stored in `1`.

Using for-loops with dictionaries

- Iterates code for each key in a dictionary.
- You create a loop variable (or iteration variable) to stand in for each key in the dictionary; e.g. the following code prints each key name stored in `my_dictionary` (but you could use any word in place of `item`):
  ```
  for item in my_dictionary:
        print(item)
  ```
- To access the values, you use the dictionary name followed by the loop variable; e.g. the following code prints each value stored in `my_dictionary`:
  ```
  for item in my_dictionary:
        print(my_dictionary[item])
  ```

The .items() method

- Returns each key-value pair in a dictionary as a tuple.

Changing dictionary items

- Access a dictionary item (using the dictionary name and the key name in square brackets) and assign a new value using the equal sign; e.g: the following code updates the value of `"first_key"` to 42:
  ```
  my_dictionary["first_key"] = 42
  ```

Adding dictionary items

- Add a dictionary item (using the dictionary name and the new key name in square brackets) and assign a new value using the equal sign; e.g: the following code would add a new key-value pair to the end of a dictionary:
  ```
  my_dictionary["new_key"] = 42
  ```

Removing dictionary items (del command or .pop() method)

- Use the `del` command with the dictionary and key name to delete a key-value pair; e.g.:
  `del my_dictionary["first_key"]`
- Or use the `.pop()` method on the dictionary name, with the key name in the parenthesis. This not only removes the key-value pair, but also returns the value; e.g. the following would remove the key-value pair stored in `"first_key"` from the dictionary, and return its value:
  `my_dictionary.pop("first_key")`

Data structures

- Combine or nest different complex data types; e.g. a list of lists or a list of dictionaries.
- Items are accessed by chaining square brackets together with the index number or key name; e.g. to access the first key of the first dictionary in a list of dictionaries, you would use:
  `my_list_of_dictionaries[0]["first_key"]`