Python Reference Guide

Background

Python is a powerful, portable, object-oriented programming language. Featuring an easy-to-learn syntax, this programming language (and its tools) is available on all major platforms. It can be used for almost everything in many different scales, including Google's search engine, YouTube, and it also powers the New York Stock Exchange (NYSE).

Check out this link for a deeper look into the language.

Python Cheat Sheet

The purpose of this sheet is to function as a quick-reference guide for many of the more common tools used within the language.

`if`, `elif`, and `else`

An `if statement` is programming logic: a conditional that will execute a part of code if the condition is met.

Syntax Notes: Directly following each conditional statement is a colon, followed again by a new indented line. The code will not run without this specific structure. It is Python convention to indent by four spaces.

```
# Save a variable
n = 35

# Write a conditional that will print a message if met
if n > 25:
    print("Greater than 25")
```

```
Greater than 25
```

Now let's break this down.

- First, a number of 35 is assigned to a variable named n `n`.
- Next, we create an `if statement` stating that if `n` is greater than 25, print "Greater than 25" to the console.
- Notice that since the conditional statement (35 is greater than 25) has been met, the print statement was generated.

When multiple conditions exist, an `if statement` can be partnered with `elif` (else if) and `else`.

```
# Save a variable
n = 15

# Write three conditionals for the code to check
if n > 25:
    print("Greater than 25")
elif n >= 20:
    print("Greater than or equal to 20")
else:
    print("Less than 25")
```

```
Less than 25
```

Expanding upon the simple `if statement` earlier, now we have an `if statement` with two additional statements: `elif` and `else`.

The code checks the variable against each statement before moving to the next, skipping the inner print statement. When the conditional is satisfied, the code outputs the inner print statement and ceases to run.

For loops

A `for loop` in Python is versatile, simple, and flexible. When a `for loop` is used, it cycles through a section of code a specified number of times.

```
for n in range(5):
    print(n)
```

```
0
1
2
3
4
```

In the above example, `n` is a variable that will increment itself at the end of each loop. Next, the word `in` is a specific keyword within the `for loop` syntax; it specifies the starting end ending points of the loop, otherwise known as the `range`.

Notice the printed output. Python, similar to many other programming languages, starts counting at `0`. It is also important to note that Python will stop counting before it reaches the end of the range. For example, `in range(5):` means that it will count up to, but not include, 5.

Lists

Lists are another tool used widely throughout Python, commonly used to hold data. They allow more than one value to be assigned to a `variable`. These values can be accessed and manipulated through various methods, making lists very flexible and powerful.

```
animals = ["dog", "cat", "rabbit"]

print(animals)
```

```
['dog', 'cat', 'rabbit']
```

In the above example, the variable `animals` is assigned multiple values within `square brackets`; these brackets tell us that we are working with a list.

The print statement returns all values within the list.

`List indexing` is a method to call particular values, or items, from the list.

```
animals = ["dog", "cat", "rabbit"]

print(animals[2])
```

```
rabbit
```

To print a single animal from the list, note that inside the print statement the variable is followed by a number within square brackets.

Each value within the list is assigned an index number. By specifying an index number, we are able to choose a single value from the list.

`.append()` is a method used to add a value to an existing list. Let's add another animal to our `animals` list.

```
animals.append("hamster")
print(animals)
```

```
['dog', 'cat', 'rabbit', 'hamster']
```

Dictionaries

Dictionaries provide another way to hold data. Whereas a value in a list is accessed by its position, a value in a dictionary is accessed by its key. The ordering of values matters in lists, but it does not matter in dictionaries.

Instead of using brackets to create it, curly braces are used.

The `keys` of a dictionary are enclosed within single or double quotes, followed by a colon then the `key value`. Each `key` and `value` pair are separated by a comma, and all pairs reside within a set of curly braces.

```
# Creating a dictionary
birds_sighted = {'dove': 3, 'chickadee': 5, 'hawk': 1}

# Printing the dictionary
print(birds_sighted)
```

```
{'dove': 3, 'chickadee': 5, 'hawk': 1}
```

```
# Printing a single value using a dictionary key
print(birds_sighted['hawk'])
```

```
1
```

Functions

Lists and dictionaries hold data. Functions do things with data. They are also extremely useful in saving code that will be used again later. This way, you won't need to re-write the same block of code repeatedly.

A `function` is defined by using the keyword `def` before the function name, followed by a set of parenthesis and a colon.

The print statement, on the next indented line, acknowledges that the function is being called.

```
def say_hello():
    print("Hello!")
```

To call the `function`, type the `function` name followed by a set of parenthesis on a new line.

```
say_hello()
```

```
Hello!
```

A `function` can also can also act on data input by a user, called an argument.

```
name = "Bob"

def say_hello(name):
    print(f"Hello, {name}!")
```

```
say_hello(name)
```

```
Hello, Bob!
```

`.csv Files`

When working in Python, you will often find yourself importing data from different sources, such as .csv files. During class, you will be importing .csv files with `os.path.join()`.

This method of reading in .csv files works on Windows and Mac operating systems, allowing you to access the saved data using Python code.

Reading a .csv file

```
# Import the necessary dependencies for os.path.join()
import os
import csv

# Read in a .csv file
csv_file = os.path.join("folder_name", "file.csv")
```

Notice that there are no back- or forwardslashes in the filepath. Using `os.path.join()` nullifies the hindrance of cross-platform bugs when reading in .csv files.

Writing a .csv file

```
# Import necessary dependencies
import csv

# Create the path for the filename
data_output = os.path.join("folder_name", "data.csv")

# Write data to a .csv file
with open(data_output, "w", newline="") as csvfile:
    writer = csv.writer(csvfile)
    # To save specific data input as a row in the csv
    writer.writerow(["row1", "row2"])
```

To learn more about reading and writing .csv files, visit the Python documentation webpage.