**Python Notes:**

[Section 1: Python Language Overview 2](#_n9topsprcpak)

[Variable assignment: 2](#_ogqsj7chiz7w)

[Variable Name Rules: 2](#_xj5g7hv36nqk)

[Comments: 2](#_wvbexclyqx6a)

[Section 2: Basic Data Types 3](#_babtxu8s3sbg)

[Individual and collection data types 3](#_p9v4izia1vi2)

[Integers 3](#_fhk7awiwdhgt)

[Floats 3](#_tolwhj6s1155)

[Strings 4](#_nc3f0nsl3clj)

[Booleans 5](#_kq9tgexp3kti)

[Lists 5](#_conbetyory3j)

[Dictionaries 6](#_mkxyd7nd2iqt)

[Section 3: Operators 8](#_mzzjr2tuhudx)

[Operator types 8](#_v6j5p0vhp6rz)

[Arithmetic Operators 8](#_avvsio4vbum0)

[Comparison Operators 9](#_ku5npfhsnur7)

[Logical Operators 9](#_omdjgqfqshn)

[Membership Operators 9](#_msvoybotxu6x)

[Section 4: Control Structures 11](#_z0xip1fqknkp)

[If and indentation 11](#_nxwtv0xjatcs)

[Nesting statements 12](#_pkyz1yizkpqv)

[If, Elif and Else 12](#_rxwrr8u1hq1p)

[While loops 13](#_fafpcs4m20uc)

[Section 5: Functions 14](#_545wk4yg1zch)

[Basic Functions 14](#_6vpee2qn1f7u)

[Arguments 14](#_npazptjef2yz)

[Return Values 15](#_z5vaz1fmb5be)

[Default Arguments 16](#_h56eh4ub6k2b)

[Section 6: Data Types Details 17](#_nnwewa1e0d7)

[Integers and Floats 17](#_2hx6jrvmdxzl)

[Strings 18](#_5ol6um14b3hb)

[Split 18](#_7ti1mzc3don)

[Replace 19](#_h2jec562dhoj)

[Slicing 19](#_uu9lq9st9a6u)

[Format 20](#_at6x6hfoeec6)

[Booleans 20](#_3l95x6k62xpe)

[Lists 21](#_dcqvc1fulyp5)

[Dictionaries 22](#_1b32eul69nxl)

[Section 7: Other Common Tasks 24](#_9v7hlymt9hhk)

[Importing 24](#_k40ba7buakee)

[Importing from the Python Standard Library 25](#_ql4gkfpm1nd7)

[User Input 27](#_psu8sug1mtq6)

[Error Handling 29](#_1bfb9n7y9uds)

# Section 1: Python Language Overview

## Variable assignment:

To assign the value of 1 to the variable “x” use the following code: *x=1*

## Variable Name Rules:

1. Must start with a letter or an underscore
2. Can only contain alphanumeric characters and underscores (A-Z, a-z, 0-9 and \_)
3. Case sensitive

Example: All the variables below look similar but when referenced would have to be referenced exactly.

*food = ‘Pasta’*

*Food = ‘Macaroni’*

*F00D = ‘Pene’*

*\_food = ‘Lasagna’*

*Food1 = ‘Fettuccine’*

**print** - prints a variable or a “string” in the immediate/Shell window, so to print the variable “x” use: ***print(****x****)***

## **Comments:**

Are text within code purely for commentary. They are not run by the program/interpreter. There are two ways to comment on code:

* Line comments - Comment on a single line of code or on a simple function that can be described in a single line. To start a “line comment” use a “#”. They can be on a line of their own or can follow a line of code.
* Multi-line comments - Used to describe a longer function or a block of code, need more than 1 line. To start a “multi-line comment” use either 3”s or 3’s:

| *“””*  *This is a multi-line comment*  *This is the 2nd line*  *Third line*  *”””* | ⇔ | *‘’’*  *This is a multi-line comment*  *This is the 2nd line*  *Third line*  *’’’* |
| --- | --- | --- |

# Section 2: Basic Data Types

## Individual and collection data types

| Individual data types   * Integers * Floats * Strings * Booleans | Collection data types:   * Lists * Dictionaries |
| --- | --- |

## Integers

Integers are whole numbers. (Kids in a school)

| *#Assigns integer to x*  *x=4*  *#prints the value of x*  ***print(****x****)*** | Shell window:  *4* |
| --- | --- |

How do we know Python is storing *“x”* as an integer? Use the ***type()*** function. The ***type()*** function tells you what the data type is. Here whatever follows “class” is the data type

| *#Assigns integer to x*  *x=4*  *#prints the value of x*  ***print(****x****)***  *#prints the datatype of x*  ***print(type(****x****))*** | Shell window:  *4*  *<class 'int'>* |
| --- | --- |

## Floats

To represent quantities that aren’t whole numbers use **floats** instead(Length of arm in cm) can be positive or negative.

| *#Assign float to x*  *x=4.01*  ***print(****x****)***  ***print(type(****x****))*** | Shell window:  *4.01*  *<class 'float'>* |
| --- | --- |

**Floats** can also be represented by scientific notation as well. To do this a lower case or capital e/E is placed after the first **float** or **integer** the following the e/E is another **integer**

| *#Assign Scientific notation float to x*  *x=3e3*  ***print(****x****)***  ***print(type(****x****))*** | Shell window:  *3000.0*  *<class 'float'>* | *#Assign Scientific notation float to z*  *z=4.01e5*  ***print(****z****)***  ***print(type(****z****))*** | Shell window:  *401000.0*  *<class 'float'>* |
| --- | --- | --- | --- |
| *#Assign Scientific notation float to y*  *y=4E4*  ***print(****y****)***  ***print(type(****y****))*** | Shell window:  *40000.0*  *<class 'float'>* |
|

## Strings

**Strings** are non numeric pieces of data (names, addresses,...) Like with comments Python accepts either single quotes (‘) or double quotes(“) for defining a string.

| *x=’Single quoted string’*  ***print(****x****)***  ***print(type(****x****))*** | Shell window:  *Single quoted string*  *<class 'str'>* | *y=”Double quoted string”*  ***print(****y****)***  ***print(type(****y****))*** | Shell window:  *Double quoted string*  *<class 'str'>* |
| --- | --- | --- | --- |

As with comments, strings also have multi-line variations. New lines and tabs are retained within a string.

| *x=’’’This is a multi-line*  *String, created with triple single quotes*  *There should be a blank row above this*  *and a tab to the left’’’*  ***print(****x****)***  ***print(type(****x****))*** | Shell window:  *This is a multi-line*  *String, created with triple single quotes*  *There should be a blank row above this*  *and a tab to the left*  *<class 'str'>* |
| --- | --- |

There is also another way to define a string in Python. Using the “string constructor” (***str(...)***). This casts non string data types to strings

| *#Create a float*  *x=1.2*  ***print(****x****)***  ***print(type(****x****))*** | Shell window:  *1.2*  *<class 'float'>* | *#Convert the float to a string*  *y =* ***str(****x****)***  ***print(****y****)***  ***print(type(****y****))*** | Shell window:  *1.2*  *<class 'str'>* |
| --- | --- | --- | --- |

## Booleans

In programming you often need to know if an expression is True or False. You can evaluate any expression in Python, and get one of two answers, True or False.

| *has\_children = True*  ***print(****has\_children****)***  ***print(type(****has\_children****))***  *signed\_up = False*  ***print(****signed\_up****)***  ***print(type(****signed\_up****))*** | Shell window:  *True*  *<class 'bool'>*  *False*  *<class 'bool'>* |
| --- | --- |

When you compare two values, the expression is evaluated and Python returns the **Boolean** answer

| *x=1*  *y=2*  *bool\_1 = x == y*  ***print(****bool\_1****)***  ***print(type(****bool\_1****))***  *z=1*  *bool\_2 = x == z*  ***print(****bool\_2****)***  ***print(type(****bool\_2****))*** | Shell window:  *False*  *<class 'bool'>*  *True*  *<class 'bool'>* |
| --- | --- |

## Lists

A **list** is a list of items, items can be **numbers, strings, floats, booleans** and even **lists**.

| *list\_of\_numbers = [1, 2, 3]*  ***print (****list\_of\_numbers****)***  ***print(type(****list\_of\_numbers****))*** | Shell window:  *[1, 2, 3]*  *<class 'list'>* |
| --- | --- |

Here is a **list** with more complex items.

| *list\_of\_stuff = [1, ‘item’. 1.2, True, [‘list’, ‘in’, ‘list’]]*  ***print (****list\_of\_stuff****)***  ***print(****type(list\_of\_stuff****))*** | Shell window:  *[1, ‘item’, 1.2, True, [‘list’, ‘in’, ‘list’]]*  *<class 'list'>* |
| --- | --- |

One method to add an item to a **list** is to use the ***append()*** function. The syntax of the append function is as follows. *list.****append(****item\_to\_add****)****,* where “*list*” is a list. ***append()*** always adds to the end of the list. To choose where to add to a list click [here](#ohhgv5csihh5)

| *number\_list = []*  *number\_list.append(1)*  ***print(****number\_list****)***  *number\_list.append(2)*  ***print(****number\_list****)*** | Shell window:  *[1]*  *[1,2]* |
| --- | --- |

To retrieve an item from a list, we use the item index. The item index is placed in [] after the list, but within the ***print()*** function. Python is zero!!! indexed this means the first item in a collection has index 0, then second item has index 1,...

| *list\_of\_things = [‘1st\_item’, ’2nd\_item’, ‘3rd\_item’]*  ***print(****list\_of\_things[1]****)*** | Shell window:  *2nd item* |
| --- | --- |

## 

## Dictionaries

Dictionaries are used to store data values in **key:value** pairs which are separated by a “:”. A dictionary is a collection which is ordered, changeable and does not allow duplicates. For example the first entry's **key** is *first\_name* and the first entry's **value** is *Tony.* Keys are typically strings but Python allows integers and floats. **keys** must be unique. **Values** can be any data type.

| *person\_details = {*  *‘first\_name’: ‘Tony’,*  *‘last\_name’ : ‘Macaroni’,*  *‘age’: 26,*  *‘job’: ‘Pasta Manufacturer’,*  *‘state’: ‘PA’, }*  ***print(****person\_details****)***  ***print(type(****person\_details****))*** | Shell window:  *{‘first\_name’: ‘Tony’, ‘last\_name’: ‘Macaroni’, ‘age’ =: 26, ‘job’: ‘Pasta Manufacturer’, ‘state’: ‘PA’}*  *<class 'dict'>* |
| --- | --- |

To access a specific value, the pairing key needs to be put in “[]” after referencing the dictionary. The same syntax is used to add key:value pairs to a dictionary

| *person\_details = {*  *‘first\_name’ : ‘Tony’,*  *‘last\_name’ : ‘Macaroni’,*  *‘age’ : 26,*  *‘job’ : ‘Pasta Manufacturer’,*  *‘state’ : ‘PA’, }*  ***print(****person\_details[‘first\_name’]****)***  ***print(****person\_details[‘last\_name’]****)*** | Shell window:  *Tony*  *Macaroni* |
| --- | --- |

Example: You want to keep track of all your friends' data, names and age.  
Solution: Create a **dictionary** for each person then store all those dictionaries in a **dictionary**. Here person\_1, person\_2 and person\_3 and friends are **dictionaries**. The friends **dictionary** is made by using the referencing method shown on the previous page. The **key** is the first name referenced and the **value** is the **dictionary** of the person

| *person\_1 = {*  *'first\_name': 'Tony',*  *'last\_name': 'Macaroni',*  *'age': 26,}*  *person\_2 = {*  *'first\_name': 'Betsy',*  *'last\_name': 'Bolognese',*  *'age': 34,}*  *person\_3 = {*  *'first\_name': 'Peter',*  *'last\_name': 'Pepperoni',*  *'age': 24,}*  *friends = {*  *person\_1['first\_name']: person\_1,*  *person\_2['first\_name']: person\_2,*  *person\_3['first\_name']: person\_3 }*  ***print(****friends****)***  *peter = friends['Peter']*  ***print(****peter['last\_name']****)*** | Shell window:  *{'Tony': {'first\_name': 'Tony', 'last\_name': 'Macaroni', 'age': 26}, 'Betsy': {'first\_name': 'Betsy', 'last\_name': 'Bolognese', 'age': 34}, 'Peter': {'first\_name': 'Peter', 'last\_name': 'Pepperoni', 'age': 24}}*  *Pepperoni* |
| --- | --- |

This is quicker than using lists for a computer as when a **list** is used the computer must count the number of indexes specified. But with a **dictionary** the computer can access by name without having to look at all other entries.

# Section 3: Operators

## Operator types

| * Arithmetic +, -, \*, /, ^, % * Comparison =, !=, >, <, >=, <= | * Logical and, or, not * Membership In, Not In |
| --- | --- |

## Arithmetic Operators

Here is a table on how to do all the arithmetic operators in Python. You can have unlimited arguments within the ***print()*** statement. It will print the arguments side by side. That is how we have complete equations in the far right hand side box.

| addition  subtraction  multiplication  division  exponent  modulus | +  -  \*  /  \*\*  % | *sum = 1 + 2*  ***print(****'1 + 2 =', sum****)***  *difference = 2 - 1*  ***print(****'2 - 1 =', difference****)***  *product = 2 \* 3*  ***print(****'2 \* 3 =', product****)***  *quotient = 6 / 3*  ***print(****'6 / 3 =', quotient****)***  *power = 3 \*\* 3*  ***print(****'3 \*\* 3 =', power****)***  *modulus = 10 % 3*  ***print(****'10 % 3 =', modulus****)*** | Shell window: *1 + 2 = 3*  *2 - 1 = 1*  *2 \* 3 = 6*  *6 / 3 = 2.0*  *3 \*\* 3 = 27*  *10 % 3 = 1* |
| --- | --- | --- | --- |

## 

## Comparison Operators

| equal  Not equal  Greater than  Less than  Greater than or equal to  Less than or equal to | ==  !=  >  <  >=  <= | *# Equal*  ***print('****1 == 1', 1 == 1****)***  *# Not equal*  ***print(****'1 != 2', 1 != 2****)***  *# Greater than*  ***print(****'2 > 1', 2 > 1****)***  *# Less than*  ***print(****'1 < 2', 1 < 2****)***  *# Greater than or equal to*  ***print(****'2 >= 1', 2 >= 1****)***  *# Less than or equal to*  ***print(****'1 <= 2', 1 <= 2****)*** | Shell window: *1 == 1 True*  *1 != 2 True*  *2 > 1 True*  *1 < 2 True*  *2 >= 1 True*  *1 <= 2 True* |
| --- | --- | --- | --- |

## Logical Operators

Logical operators are used to combine conditional statements.

| Operator | Description | Example code | Shell window: |
| --- | --- | --- | --- |
| and | Returns True if both statements are true | *x = 5*  ***print(****x > 3 and x < 10****)*** | *True* |
| or | Returns True if one of the statements is true | *x=5*  ***print(***x >3 or x < 4**)** | *True* |
| *x=2*  ***print(***x >3 or x < 4**)** | *True* |
| not | Reverse the result, returns False if the result is true | *x = 5*  ***print(not(****x > 3 and x < 10****))*** | *False* |

## Membership Operators

Membership operators are used to test if an item exists in a collection. That item can be anything. You can search for lists, within lists.

| Operator | Description | Example code | Shell window: |
| --- | --- | --- | --- |
| In | Returns True if an item with the specified value is present in the collection | *items = [1,2,3,4]*  ***print(****4 in items****)*** | *True* |
| Not In | Returns True if an item with the specified value is not present in the collection | *items = [1,2,3,4]*  ***print(****5 not in items****)*** | *True* |

When using **membership** operators with **dictionaries** the logic is a bit different. The **In** operator only checks the **keys** not the **values**!

| *person = {*  *'first\_name': 'Tony',*  *'last\_name': 'Macaroni',*  *'age': 29*  *}*  ***print(****'first\_name in person?', 'first\_name' in person****)***  ***print(****'Tony in person?', 'Tony' in person****)*** | Shell window:  *first\_name in person? True*  *Tony in person? False* |
| --- | --- |

# 

# Section 4: Control Structures

| * Conditional statements   + If   + Elif   + Else | * Loops   + While   + For |
| --- | --- |

## If and indentation

The ***if*** statement is a conditional statement. It is used to execute a block of code only when a specific condition is met. The statement also always ends with a colon(:)

| *a = 33*  *b = 200*  ***if*** *b > a:*  ***print(****"b is greater than a"****)***  *# Notice the indentation on the row above.* | Shell window:  *b is greater than a* |
| --- | --- |

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code.

| *a = 33*  *b = 200*  ***if*** *b > a:*  ***print(****"b is greater than a"****)***  *# No indentation will lead to an error.* | Shell window:  ***print(****"b is greater than a"****)***  *^*  *IndentationError: expected an indented block after 'if' statement on line 3* |
| --- | --- |

Boolean and multiple if example:

| *i = 1*  *# Using an ‘or’ statement so only 1 has to be true for the print to execute.*  ***if*** *i == 'a' or i == 1:*  ***print(****'statement\_1'****)***  *# Second if statement says if true print statement\_2, since true is already true the statement will be executed.*  ***if*** *True:*  ***print(****'statement\_2’****)*** | Shell window:  *statement\_1*  *statement\_2* |
| --- | --- |

## Nesting statements

Putting a ***if*** statement inside another ***if***statement is called nesting. This is done by further indentation.

| *i = 1*  ***if*** *i == 'a' or i == 1:*  ***print(****'statement\_1'****)***  ***if*** *True:*  ***print(****'statement\_2’****)***  *# The second if statement is indented within the first if statement.* | Shell window:  *statement\_1*  *statement\_2* |
| --- | --- |

If ‘i’ is set to 2, the first ***if*** statement will return nothing due to ‘2’ is equal to ‘a’ or ‘1’ and therefore the second ***if*** statement will not execute. If this was done without them being nested the second ***if*** statement would run no matter the result of the 1st ***if*** statement.

| *i = 2*  ***if*** *i == 'a' or i == 1:*  ***print(****'statement\_1'****)***  ***if*** *True:*  ***print(****'statement\_2’****)***  *# The second if statement is indented within the first if statement. Nothing in the shell window.* | Shell window: |
| --- | --- |

## If, Elif and Else

If you want to do something different when the ***if*** statement is false, you can use both ***elif***and ***else*** to execute something else*.* ***Elif*** is another ***if*** statement that will execute if the previous conditions were not true. Then ***else*** is used if neither the original ***if*** or any of the ***elif***’s are true. ***Elif*** can be used as many times as needed.

In this example ‘a’ is greater than ‘b’, so the first ***if*** is False, also the ***elif*** statement is False, so the code executes the ***else*** statement and prints "a is greater than b".

| *a = 200*  *b = 33*  ***if*** *b > a:*  ***print(****"b is greater than a"****)***  ***elif*** *a == b:*  ***print(****"a and b are equal"****)***  ***else****:*  ***print(****"a is greater than b"****)*** | Shell window:  *a is greater than b* |
| --- | --- |

## 

## While loops

The ***while*** loop executes a set of statements as sling as a condition is true. It stops when the condition is False. If the condition never becomes false the while loop will run forever. You can stop code running in Pycharm by pressing the red button or using the shortcut ‘ctrl’ + ‘c’

| *i = 1# Sets i to 1*  ***while*** *i < 6: # Checks that the condition is true*  ***print(****'The current count is',i****)***  *i += 1# Increments the value of i by adding 1* | Shell window:  *The current count is 1*  *The current count is 2*  *The current count is 3*  *The current count is 4*  *The current count is 5* |
| --- | --- |

## 

# Section 5: Functions

Functions are named pieces of code that only run when they’re called.

* Basic Functions
* Function Arguments - Passing arguments into a function
* Return Values - Returning values from a function
* Default Arguments - Set default arguments for functions.

## Basic Functions

To define a function in Python the word ***def*** is used followed by the name of the function and a set of brackets. At the end of the definition a ‘:’ is used. Everything indented under the function will be run when the function is called. The order in which functions are determined is irrelevant; it only matters what order the functions are called.

| *# define the function name*  *def print\_message():*  *# code the function executes*  ***print(****'Hello from within a function!'****)***  *# Call the function so it executes*  *print\_message()* | Shell window:  *Hello from within a function!* |
| --- | --- |

## Arguments

An argument is the value that is sent to the function when it is called.

Example: You want to design a function that takes a person’s full name and splits it up into first name and last name.  
Solution: The example below does make a bad assumption that all names passed into the function will be represented by ‘first name’ ‘(space)’ ‘Last name’.

| *# Any value that gets passed into the function will be assigned to the variable ‘full\_name’.*  ***def*** *first\_last(full\_name):*  *first\_name = '' # store the letters of the first name.*  *last\_name = '' # store the letters of the last name.*  *has\_been\_a\_space = False # This is a boolean initially set to False but will change to True once a space has been encountered.*  ***for*** *letter* ***in*** *full\_name: # This for loop iterates over each letter in the full name variable.*  ***if*** *letter == ' ':*  *has\_been\_a\_space = True*  ***elif*** *has\_been\_a\_space:*  *last\_name = last\_name + letter*  ***else****:*  *first\_name = first\_name + letter*  *# The loop adds all the letters to the ‘first\_name’ variable until it come across a ‘ ‘ then the boolean value is changed to True and all further letters are added to the ‘last\_name’ variable.*  ***print(****'First: ', first\_name****)***  ***print(****'Last: ', last\_name****)***  ***first\_last('****Tony Macaroni'****)*** | Shell window:  *First: Tony*  *Last: Macaroni* |
| --- | --- |

## Return Values

To let a function return a value, use the ***return*** statement. ***Return*** tells Python to stop running the function and to pass any values that are beside it back to the place the function was called from.

Example: This function takes an <int> test score and returns a teacher comment based on the score.

| ***def*** *teacher\_comment(test\_score):*  ***if*** *test\_score > 90:*  ***return*** *'Great Job'*  ***elif*** *test\_score > 80:*  ***return*** *'Pretty good'*  ***else****:*  ***return*** *'You could do better'*  ***print(****This comment will never be printed****)***  *# Notice how the final print argument was never executed as it is located after a return function.*  ***for*** *score* ***in*** *[95, 85, 75]:*  *comment = teacher\_comment(score)*  ***print(****score, '-->', comment****)*** | Shell window:  *95 --> Great Job*  *85 --> Pretty good*  *75 --> You could do better* |
| --- | --- |

## Default Arguments

Let’s use the teacher feedback example from above. A second argument has been added to the *‘teacher\_comment’.* As well as ‘test\_score’, ‘student\_name’ has been added it has also been set equal to ‘ ‘ which is an empty string. This ‘ ‘ is the default argument, so if only a ‘test\_score’ and **NO** ‘student\_name’ is passed through the function will still work and it will just use the empty string as the students name. When a value for ‘student name is passed through the function this value will override the empty string.

| *def teacher\_comment(test\_score, student\_name=' '):*  *# This function takes an <int> test score and returns a teacher comment based on the score*  *# adding student name is optional.*  ***if*** *test\_score > 90:*  *return 'Great Job ' + student\_name*  ***elif*** *test\_score > 80:*  *return 'Pretty good ' + student\_name*  ***else****:*  *return 'You could do better ' + student\_name*  *# Test scores area dictionary, keys are student names and scores are the values.*  *students = {'Tony': 95, 'Betsy': 85, 'Peter': 75}*  *# Iterate over the dictionary, each iteration pulls the score from the dictionary.*  *# The comment variable is made by running the score and name(student) in the teacher\_comment function.*  ***for*** *student* ***in*** *students:*  *score = students[student]*  *comment = teacher\_comment(score, student)*  ***print(****score, '-->', comment****)***  *#Test case with no name to prove default argument works.*  *no\_name = teacher\_comment(40)*  ***print(****'No Name 40 -->', no\_name****)*** | Shell Window:  *95 --> Great Job Tony*  *85 --> Pretty good Betsy*  *75 --> You could do better Peter*  *No Name 40 --> You could do better* |
| --- | --- |
| *# without the default argument the code errors*  *def teacher\_comment(test\_score, student\_name):*  *…*  *…*  *no\_name = teacher\_comment(40)*  ***print(****'No Name 40 -->', no\_name****)*** | Shell Window:  *TypeError: teacher\_comment() missing 1 required positional argument: 'student\_name'* |

## 

# Section 6: Data Types Details

This section will expand on the details of the data types previously shown/discussed.

| * Integers and floats * Strings | * Booleans * Dictionaries |
| --- | --- |

## Integers and Floats

The example below demonstrates how a string can be cast to different data types. Casting was briefly mentioned [here](#cxpl0dfl2qrr). The functions ***str()***, ***int()***  and  ***float()*** are used to cast one data type into another.

| *# the number 1 is set as a string*  *one\_string = '1'*  *# casts the string into an integer and a float*  *one\_string\_to\_int =* ***int(****one\_string****)***  *one\_string\_to\_float =* ***float(****one\_string****)***  *# prints all three as well as their data type*  ***print(****one\_string,* ***type(****one\_string****))***  ***print(****one\_string\_to\_int,* ***type(****one\_string\_to\_int****))***  ***print(****one\_string\_to\_float,* ***type(****one\_string\_to\_float****))*** | Shell Window:  *1 <class 'str'>*  *1 <class 'int'>*  *1.0 <class 'float'>* |
| --- | --- |

To cast an integer to a string the string needs to represent an integer. You **can’t** cast the string ‘1.2’ to the integer ‘1’. Here is a more detailed example of this.

| *a =* ***int(****2.8****)*** *# Float can be cast to an integer but python will just ignore everything after the decimal point.(Truncate)*  *b =* ***int(****'2'****)*** *# String can be cast to an integer.*  *# int('2.8') is not supported, can’t cast a decimal string.*  *c =* ***float(****'2'****)*** *# String can be cast to a float.*  *d =* ***float(****'2.1'****)*** *# Decimal strings can be cast to a float.*  *e =* ***float(****2****)*** *# Integers can be cast to floats.*  *f =* ***int(float(****'2.8'****))*** *# How to cast from a decimal string to an integer. The string must be cast to a float first. (Truncate)*  ***print(****a,* ***type(****a****))***  ***print(****b,* ***type(****b****))***  ***print(****c,* ***type(****c****))***  ***print(****d,* ***type(****d****))***  ***print(****e,* ***type(****e****))***  ***print(****f,* ***type(****f****))*** | Shell Window:  *2 <class 'int'>*  *2 <class 'int'>*  *2.0 <class 'float'>*  *2.1 <class 'float'>*  *2.0 <class 'float'>*  *2 <class 'int'>* |
| --- | --- |

Python by default truncates when casting decimal strings and floats to integers. To round the values instead the ***round()*** function is used.The ***round()*** function returns a floating point number that is a rounded version of the specified number, with the specified number of decimals. The ***round()***  function has 2 arguments. The first is the number that is to be rounded. The second argument is the number of decimal places to round the first argument. The default argument for this is 0.

| *a = 1.234567890*  *b = 1.8*  *# 2nd argument not used so default of 0 is used.*  ***print(****'round(a) -->',* ***round(****a****))***  ***print(****'round(b) -->',* ***round(****b****))***  *# These will be rounded to 2dp.*  ***print(****'round(a, 2) -->',* ***round(****a, 2****))***  ***print(****'round(b, 2) -->',* ***round(****b, 2****))*** *# This will not appear as 1.80 as*  *# Python knows not to add implied precision to a decimal value.* | Shell Window:  *round(a) --> 1*  *round(b) --> 2*  *round(a, 2) --> 1.23*  *round(b, 2) --> 1.8* |
| --- | --- |

***Round()*** does not support rounding to positions left of the decimal point. You **can’t** round to the nearest tens or hundreds.

## Strings

There are loads of functions that can be used with strings, here are some of them:

| * split() * Slicing strings(string[]) | * replace() * format() |
| --- | --- |

### Split

The string.***split()*** is a function that splits a string into a list where each word is a list item. The ***split()*** has 2 arguments, both of which are optional. The first is the separator(The character/s that will split the function on). The default for this is any white space separator. The second argument is how many splits. The default value for this is ‘-1’. Which are all occurrences.

Earlier there was an [example](#qjnjiaq5qlf7) that split first and last names by using a **for**loop. Ths code was not very efficient and needed the assumption that all names would be entered ‘first\_name’(space)’last\_name’. The split function is a far more appropriate way to achieve the same goal.

| *name = 'Tony Edward Macaroni'*  *# Only the first argument is used in this case*  *names = name.****split(****' '****)***  ***print(****names****)*** | Shell Window:  *['Tony', 'Edward', 'Macaroni']* |
| --- | --- |

### Replace

The string.***replace()*** function replaces a specified phrase with another specified phrase. The function has three arguments. The first two are required and the third is optional. The first argument is ‘oldvalue’ ; this is the string to be searched for. The second argument is ‘newvalue’ ; this is the string to replace the ‘oldvalue’ with. The final optional argument is ‘count’ ; this is A number specifying how many occurrences of the old value you want to replace. Default is all occurrences.

| *sentence = 'This awesome spaghetti is awesome'*  *better\_sentence = sentence.****replace(****'awesome', 'fabulous'****)***  ***print(****sentence****)***  ***print(****better\_sentence****)***  *date = '12/01/2035'*  ***print(****date****)***  ***print(****date.****replace(****'/', '-'****))*** | Shell Window:  *This awesome spaghetti is awesome*  *This fabulous spaghetti is fabulous*  *12/01/2035*  *12-01-2035* |
| --- | --- |

### Slicing

Slicing a string is when you want to return a specific range of characters from a string. To slice a string use the following syntax: string followed by square brackets( string[...] ).There are multiple ways to select specific characters as seen below in the comments of the code:

| *# <string>[0] returns the character at index zero in the string.*  *# <string>[-1] returns the last character in the string.*  *# <string>[:3] returns the first 3 characters in the string, from index 0 to index 2.*  *# <string>[3:] returns all but the first 3 characters in the string, from index 3 on.*  *# <string>[3:5] returns all from index 3 to index 5 of the string.*  *# <string>[3:-2] returns all from index 3, to the second to last index of the string.*  *numbers = '1234567890'*  ***print(****'numbers[0]', numbers[0]****)***  ***print(****'numbers[-1]', numbers[-1]****)***  ***print(****'numbers[:3]', numbers[:3]****)***  ***print(****'numbers[3:]', numbers[3:]****)***  ***print(****'numbers[3:5]', numbers[3:5]****)***  ***print(****'numbers[3:-2]', numbers[3:-2]****)*** | Shell Window:  *numbers[0] 1*  *numbers[-1] 0*  *numbers[:3] 123*  *numbers[3:] 4567890*  *numbers[3:5] 45*  *numbers[3:-2] 45678* |
| --- | --- |

### Format

The ***format()*** method formats the specified value(s) and inserts them inside the string's placeholder. The placeholder is defined using curly brackets: {}. Here is the syntax: string.***format(***value\_1, value\_2, ...***)***. ***format()*** is useful when combining loads of different data types into a string(s). Below is an example showing how much more efficient ***format()*** is compared with a more bodge job method.

| *Name = 'Tony'*  *age = 26*  *# Easy to forget to put (space(s)) inside the strings and also convert data types before adding them.*  *sentence = 'My name is ' + name + ', I am ' + str(age) + ' years old'*  ***print(****sentence****)***  *# Using Format with the two arguments being name and age and two sets of {}.*  *sentence\_again = 'My name is {}, I am {} years old'.format(name, age)*  ***print(****sentence\_again****)*** | Shell Window:  *My name is Tony, I am 26 years old*  *My name is Tony, I am 26 years old* |
| --- | --- |

## Booleans

Here are 9 test case dictionaries. Each case has a key(description) and value(value). Then we have a ***for*** loop that iterates over each test case to check if the case is True or False. Almost any value is evaluated to True if it has some sort of content. Any number is True, except 0. Any string, list or dictionary is True, except empty ones.

| *cases = [*  *{'description': 'Empty String',*  *'value': ''},*  *{'description': 'Non-empty String',*  *'value': 'Not empty'},*  *{'description': 'Empty List',*  *'value': []},*  *{'description': 'Non-empty List',*  *'value': [1]},*  *{'description': 'Empty Dictionary',*  *'value': {}},*  *{'description': 'Non-empty Dictionary',*  *'value': {'name': 'Tony'}},*  *{'description': 'Integer Zero',*  *'value': 0},*  *{'description': 'Integer One',*  *'value': 1},*  *{'description': 'Non-zero, non-one, Integer',*  *'value': 2},]*  ***for*** *case in cases:*  ***if*** *case['value']:*  ***print(****'{} --> True'.****format(****case['description']****))***  ***else****:*  ***print(****'{} --> False'.****format(****case['description']****))*** | Shell Window:  *Empty String --> False*  *Non-empty String --> True*  *Empty List --> False*  *Non-empty List --> True*  *Empty Dictionary --> False*  *Non-empty Dictionary --> True*  *Integer Zero --> False*  *Integer One --> True*  *Non-zero, non-one, Integer --> True* |
| --- | --- |

In the example previously the ***if*** statement checked the boolean value of each. If however we specify to check for ‘True’ this will instead check explicitly if the case is equal to ‘True’. From the result the only time any of the cases printed ‘True’ was the integer 1. Every other case printed with Not. This is because the number 1 and ‘True’ are stored as a single bit in binary with a value of 1.

The same can be said the other way. If we were to change the ***if*** statement to check explicitly for ‘False’ the integer 0 would be the only value with a different print value to the others. This is because the number 0 and ‘False’ are stored as a single bit in binary with a value of 0.

| *cases = [*  *{'description': 'Empty String',*  *'value': ''},*  *…*  *{'description': 'Non-zero, non-one, Integer',*  *'value': 2},*  *]*  ***for*** *case in cases:*  ***if*** *case['value'] == True:*  ***print(****'{} --> True'.****format(****case['description']****))***  ***else****:*  ***print(****'{} --> Not'.****format(****case['description']****))*** | Shell Window:  *Empty String Not*  *Non-empty String Not*  *Empty List Not*  *Non-empty List Not*  *Empty Dictionary Not*  *Non-empty Dictionary Not*  *Integer Zero Not*  *Integer One == True*  *Non-zero, non-one, Integer Not* |
| --- | --- |

## Lists

Lists can also be sliced just like strings, they use the same syntax: list[...]

To find the length of a list the ***len()*** function is used. The list is just put inside the the ***len()*** function. ***len()*** can also be used on strings to obtain the number of characters and dictionaries to obtain the number of key-value pairs.

| *l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]*  ***print(****l[:3]****)***  *# Elements from index 0 to index 3*  ***print(****l[3:5]****)***  *# Elements at index 3 and 4*  ***print(****l[-1]****)*** *# Last element in the list*  *count =* ***len(****l****)***  ***print(****'Length of l:', count****)*** | Shell Window:  *[1, 2, 3]*  *[4, 5]*  *0*  *Length of l: 10* |
| --- | --- |

The ***insert()*** method inserts the specified value at the specified position. ***Insert()*** has two arguments both of which are required. The first is ‘pos’ which is the number specifying which position to insert the value(remember python is 0 index, start at 0!!!!!!!). The second is ‘elmnt’ which is an element of any type(string, number, object,...).

| *L1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]*  *# Inserts Tony at index 4 and shifts items after 4 to the right.*  *L1****.insert(****4, 'Tony'****)***  ***print(****L1****)***  *# Replace the value at index 5 with the string 'five'.*  *L1[5] = 'five'*  ***print(****L1****)***  *L2 = ['a', 'b', 'c']*  *# Combines L1 and L2. The order they are in is the order they are added.*  ***print(****L1 + L2****)*** | Shell Window:  *[1, 2, 3, 4, 'Tony', 5, 6, 7, 8, 9, 0]*  *[1, 2, 3, 4, 'Tony', 'five', 6, 7, 8, 9, 0]*  *[1, 2, 3, 4, 'Tony', 'five', 6, 7, 8, 9, 0, 'a', 'b', 'c']* |
| --- | --- |

To remove items from a list the ***remove()*** function is used. ***Remove()*** only has 1 argument which is ‘elmnt’ the element to be removed. This only removes the first instance of the element. To remove an element by its index rather than the element the ***pop()*** function is used. The ***pop()*** function has 1 optional argument which is a number specifying the index of the position of the element that is to be removed. The default value is ‘-1’ which is the last item.

| *L1 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']*  *# specify the element in the remove() function.*  *L1.****remove(****'c'****)***  ***print(****L1****)***  *# specify the index of the element in the pop() function.*  *L1.****pop(****2****)***  ***print(****L1****)*** | Shell Window:  *['a', 'b', 'd', 'e', 'f', 'g', 'h', 'i']*  *['a', 'b', 'e', 'f', 'g', 'h', 'i']* |
| --- | --- |

## Dictionaries

Dictionaries have no order in Python. You can’t pull items based on an index. The key value pairs are stored in whatever order Python feels like, more accurately, they aren’t stored in any order.

Adding an item to the dictionary is done by using a new index key and assigning a value to it. Modifying an existing value is done in a very similar manner, just specify the dictionary, put the key in square brackets that you want to change and set that equal to the new value that you want.

| *person = {'first\_name': 'Tony',*  *'last\_name': 'Macaroni',*  *'age': 26}*  ***print(****person****)***  *# This line adds a new key:value pair.*  *person['job'] = 'Python Developer'*  ***print(****person****)***  *# Changes the age value.*  *person['age'] = 27*  ***print(****person****)*** | Shell Window:  *{'first\_name': 'Tony', 'last\_name': 'Macaroni', 'age': 26}*  *{'first\_name': 'Tony', 'last\_name': 'Macaroni', 'age': 26, 'job': 'Python Developer'}*  *{'first\_name': 'Tony', 'last\_name': 'Macaroni', 'age': 27, 'job': 'Python Developer'}* |
| --- | --- |

Similar to lists there are two ways of removing items from a dictionary. The ***del()*** function removes the item by referencing the key name. It uses the same syntax as the previous example. The other is by using the ***pop()*** function to reference the key you want to remove. The example below shows both methods of removing items from a dictionary.

| *person = {'first\_name': 'Tony',*  *'last\_name': 'Macaroni',*  *'age': 26,*  *'job': 'Python Developer'}*  ***print(****person****)***  *# Must include ‘[]’ otherwise python will delete the whole dictionary.*  ***del*** *person['age']*  ***print(****person****)***  *# Just need key of the item to be removed.*  *person.pop('job')*  ***print(****person****)*** | Shell Window:  *{'first\_name': 'Tony', 'last\_name': 'Macaroni', 'age': 26, 'job': 'Python Developer'}*  *{'first\_name': 'Tony', 'last\_name': 'Macaroni', 'job': 'Python Developer'}*  *{'first\_name': 'Tony', 'last\_name': 'Macaroni'}* |
| --- | --- |

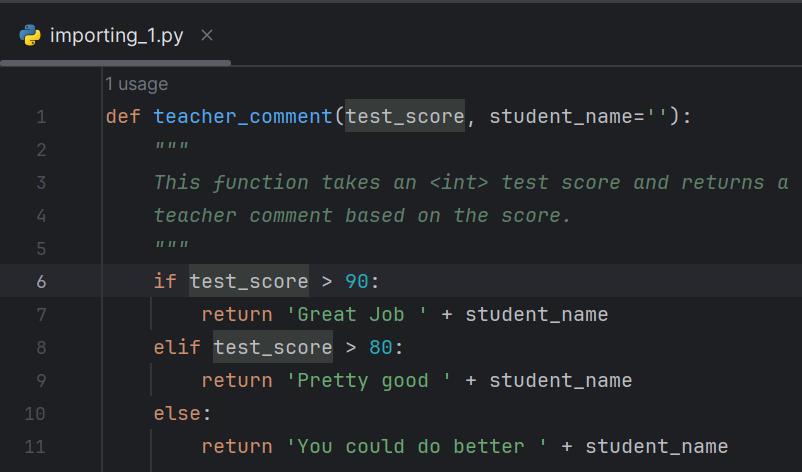
# 

# Section 7: Other Common Tasks

| * Importing * Importing from the Python standard library | * User Input * Error Handling |
| --- | --- |

## Importing

Python allows functions to be imported from other files into the one being worked on. This is done by using the ***import*** function. For example, to import the teacher feedback function into another, we first have to save the teacher feedback function. It is saved as importing\_1.py



To use this function in another function it is called it by using ***import*** followed by the file name of the function being called.

| ***import*** *importing\_1*  *students = {'Tony': 95, 'Betsy': 85, 'Peter': 75}*  *# Loop through each student's test score and print the corresponding teacher comment*  ***for*** *student in students:*  ***print(****importing\_1.teacher\_comment(students[student], student)****)*** | Shell Window:  *Great Job Tony*  *Pretty good Betsy*  *You could do better Peter* |
| --- | --- |

Some files have multiple functions within them. Importing all of it is not always needed. If only 1 function from a file is needed the syntax to ***import*** is slightly different. Example The file importing\_3 has the following 2 functions defined within it.

| * teacher\_comment | * teacher\_comment\_with\_grade |
| --- | --- |

To import only the ‘teacher\_comment\_with\_grade’ function the syntax is as follows:

***from*** ‘filename’ ***import ‘***function\_name’

If more than 1 function needs to be imported just use a ‘,’ after the first function followed by the name of the next function to be imported.

| *# The teacher comment\_with grade function also gives the score.*  ***from*** *importing\_3* ***import*** *teacher\_comment\_with\_grade*  *students = {'Tony': 95, 'Betsy': 85, 'Peter': 75}*  ***for*** *student* ***in*** *students:*  ***print(****teacher\_comment\_with\_grade(students[student], student)****)*** | Shell Window:  *Great Job Tony! You got a 95*  *Pretty good Betsy. You got a 85*  *You could do better Peter. You got a 75* |
| --- | --- |

## Importing from the Python Standard Library

Packages are typically groups of functions that are used to accomplish similar things.   
The OS module in Python provides functions for interacting with the operating system. OS comes under Python’s standard Library. This module provides a portable way of using operating system-dependent functionality. It is written generically, so it should work on any operating system.   
The Platform module is used to retrieve as much possible information about the platform on which the program is being currently executed.

Below are some examples of what can be done with ‘os’ and ‘platform’

| *# Have to use platform as well as os because os.uname() isn't implemented in Python for Windows.*  ***import*** *os*  ***import*** *platform*  *# We are using the .format() function to add the function output to our print string.*  *# This is because we don't know for sure the data type the functions are going to output.*  *user = os.****getlogin()***  ***print(****'****\n****Current Logged in User:****\n****{}'****.format(****user****))***  *# '\n' is used to represent a new line*  *info = platform.****uname()***  ***print(****'****\n****Current Operating System Info:****\n****{}'****.format(****info****))***  *files = os.****listdir()***  ***print(****'****\n****Files in Current Working Directory:****\n****{}'****.format(****files****))*** | Shell Window:  *Current Logged in User:*  *Jake*  *Current Operating System Info:*  *uname\_result(system='Windows', node='Jake\_Shiel', release='11', version='10.0.22631', machine='AMD64')*  *Files in Current Working Directory:*  *['.idea', 'error\_handling\_1.py', 'error\_handling\_2.py', 'error\_handling\_3.py', 'importing\_1.py', 'importing\_2.py', 'importing\_3.py', 'importing\_4.py', 'import\_os.py', 'import\_random.py', 'user\_input\_1.py', 'user\_input\_2.py', '\_\_pycache\_\_']* |
| --- | --- |

The first function ‘os.getlogin()’ returns the name of the operating systems user that ran the script. Another example to use this would be to black list or white list users to run a script that you have written.

The second function is ‘platform.uname()’. This returns information about the current operating system. The documentation specifies 5 pieces of data. This function could be helpful when writing a script that installs software. The right information is critical in installing a compatible version.

| * sysname - OS name * nodename - Name of machine on network * release - OS release | * version - OS version * machine - hardware identifier |
| --- | --- |

The final function is ‘os.listdir()’ returns all of the file names within a particular folder. The function takes a path to a folder as a string data type and it is an optional argument. If no path is given the function will return all of the file names in the directory that the script is run from, this directory is also known as current working directory.

Another package built within Python is called random. The random package generates random numbers in Python. These are pseudo-random numbers meaning they are not truly random. The Python doc explicitly states that due to being pseudo random it should not be used for security purposes.

This package can be used to perform random actions such as generating random numbers, printing random values for a list or string, etc.

| ***import*** *random*  *letters = ['a', 'b', 'c', 'd', 'e', 'f']*  *count = 0*  ***while*** *count < 10:*  *# Generate a random integer between 0 and 9 (inclusive)*  *random\_int = random****.randint(****0, 9****)***  *random****.shuffle(****letters****)***  *# Shuffle the order of elements in the letters list*  *choice = random****.choice(****letters****)*** *# Choose a random element from the shuffled letters list*  ***print(****'Random\_Integer: {}, Shuffle: {}, Choice: {}'.****format(****random\_int, letters, choice****))***  *count += 1* | |
| --- | --- |
| Shell Window:  *Random\_Integer: 3, Shuffle: ['a', 'e', 'b', 'd', 'f', 'c'], Choice: b*  *Random\_Integer: 2, Shuffle: ['b', 'f', 'a', 'c', 'd', 'e'], Choice: f*  *Random\_Integer: 6, Shuffle: ['f', 'd', 'a', 'e', 'b', 'c'], Choice: b*  *Random\_Integer: 9, Shuffle: ['b', 'c', 'd', 'f', 'e', 'a'], Choice: e*  *Random\_Integer: 5, Shuffle: ['b', 'c', 'f', 'd', 'e', 'a'], Choice: e*  *Random\_Integer: 0, Shuffle: ['a', 'f', 'c', 'e', 'b', 'd'], Choice: c*  *Random\_Integer: 7, Shuffle: ['d', 'c', 'b', 'f', 'a', 'e'], Choice: d*  *Random\_Integer: 1, Shuffle: ['a', 'f', 'c', 'b', 'e', 'd'], Choice: e*  *Random\_Integer: 0, Shuffle: ['b', 'c', 'd', 'f', 'a', 'e'], Choice: d*  *Random\_Integer: 7, Shuffle: ['d', 'f', 'b', 'c', 'a', 'e'], Choice: f* | |

## User Input

Often programs require the user to interact with the script. To get user input, use the ***input()*** function. It has the following syntax. ***input(****prompt****)***. ‘Prompt’ is a string that represents the default message before the user's input.

| *name =* ***input('****what is your name? '****)***  ***print(****'Hello {}!'****.format(****name****))***  *age =* ***int(input(****''How old are you? '****))***  *current\_year = 2024*  *year\_born = current\_year - age*  ***print(****'Hello {}! you were born in {} '****.format(****name,year\_born****))*** | Shell Window:  *what is your name?* |
| --- | --- |

Once the script is run the Shell Window becomes an input area where you can type your name.

| *name =* ***input('****what is your name? '****)***  ***print(****'Hello {}!'****.format(****name****))***  *age =* ***int(input(****''How old are you? '****))***  *current\_year = 2024*  *year\_born = current\_year - age*  ***print(****'Hello {}! you were born in {} '****.format(****name,year\_born****))*** | Shell Window:  *what is your name?* Jake |
| --- | --- |

Once the name is typed, press (Enter) and the script will run the next part of the script and say hello to the user. Then ask for their age.

| *name =* ***input('****what is your name? '****)***  ***print(****'Hello {}!'****.format(****name****))***  *age =* ***int(input(****''How old are you? '****))***  *current\_year = 2024*  *year\_born = current\_year - age*  ***print(****'Hello {}! you were born in {} '****.format(****name,year\_born****))*** | Shell Window:  *what is your name?* Jake  *Hello Jake!*  *'How old are you? 26* |
| --- | --- |

Once age is input the final part of the script will be run and greet the user with their age and their date of birth.

| *name =* ***input('****what is your name? '****)***  ***print(****'Hello {}!'****.format(****name****))***  *# Need to convert to integer for the calculation to work.*  *age =* ***int(input(****''How old are you? '****))***  *current\_year = 2024*  *year\_born = current\_year - age*  ***print(****'Hello {}! you were born in {} '****.format(****name,year\_born****))*** | Shell Window:  *what is your name? Jake*  *Hello Jake!*  *'How old are you? 26*  *Hello Jake! you were born in 1998* |
| --- | --- |

## 

## 

Here is a more complicated example using ***while*** and ***if*** loops. The **break** keyword is used to break out a for loop, or a while loop. The script prints a list of names then gives the user a list of options to do

| ***import*** *random*  *# Define a multiline string containing the menu options*  *menu = '''What do you want to do?*  *1) Add a name*  *2) Select a random name*  *3) Remove a name*  *q) quit*  *'''*  *names = ['Tony', 'Betsy', 'Peter']*  *# Start an infinite loop to continuously display the menu and prompt for user input.*  ***while*** *True:*  ***print(****names****)***  *choice =* ***input(****menu****)****# Prompt the user to input their choice from the menu.*    *# Check if the user wants to quit.*  ***if*** *choice.lower() == 'q':*  ***break*** *# Exit the loop and end the program.*  *# Check if the user wants to add a name.*  ***elif*** *choice == '1':*  *new\_name =* ***input(****'What name do you want to add?'****)***  *names.append(new\_name)*    *# Check if the user wants to select a random name.*  ***elif*** *choice == '2':*  ***print(****'The random name is {}****'.format(****random****.choice(****names****)))***  *# No variable is assigned to the value of input so Python waits for the user to press (Enter)*  ***input(****'Press Enter to Continue'****)***    *# Check if the user wants to remove a name*  ***elif*** *choice == '3':*  *name\_to\_remove =* ***input(****'What name do you want to remove?'****)***  *names****.remove(****name\_to\_remove****)*** | |
| --- | --- |
| Shell Window:  *['Tony', 'Betsy', 'Peter']*  *What do you want to do?*  *1) Add a name*  *2) Select a random name*  *3) Remove a name*  *q) quit* | |

## Error Handling

When an error occurs, or an exception, Python will normally stop and generate an error message. The error first states which file the error was found in. The next thing displayed is the line number where the error occurred.Then the line with the error is printed. Next the error type is shown.In the example below option 3 has been picked and a name that is not in the list has been used as an input.

| ***import*** *random*  *menu = '''What do you want to do?*  *1) Add a name*  *2) Select a random name*  *3) Remove a name*  *q) quit*  *'''*  *names = ['Tony', 'Betsy', 'Peter']*  ***while*** *True:*  ***print(****names****)***  *choice =* ***input(****menu****)***  *…….*  ***elif*** *choice == '3':*  *name\_to\_remove =* ***input(****'What name do you want to remove?'****)***  *names****.remove(****name\_to\_remove****)*** | |
| --- | --- |
| Shell Window:  *['Tony', 'Betsy', 'Peter']*  *What do you want to do?*  *1) Add a name*  *2) Select a random name*  *3) Remove a name*  *q) quit*  *3*  *What name do you want to remove?Ethan*  *Traceback (most recent call last):*  *File "C:\Users\Jake\Documents\Data Analysis\Beginner Python(Udemy)\08 Other Common Tasks\error\_handling\_1.py", line 24, in <module>*  *names.remove(name\_to\_remove)*  *ValueError: list.remove(x): x not in list* | |

In order to fix this a ***try*** and ***except*** block are used to handle errors. The ***try*** block lets you test a block of code for errors. The ***except*** block lets you handle the error. Since the ***try*** block raises an error, the ***except*** block will be executed.

| *menu = '''What do you want to do?*  *1) Add a name*  *2) Select a random name*  *3) Remove a name*  *q) quit*  *'''*  *…….*  ***elif*** *choice == '3':*  *name\_to\_remove =* ***input(****'What name do you want to remove?'****)***  ***try****:*  *# Try to remove the specified name from the list of names.*  *names****.remove(****name\_to\_remove****)***  *# The except block catches whatever exception is and assigns it to e.*  ***except*** *Exception as e:*  ***print(****e****)***  ***input(****'press Enter to continue'****)*** | |
| --- | --- |
| Shell Window:  *['Tony', 'Betsy', 'Peter']*  *What do you want to do?*  *1) Add a name*  *2) Select a random name*  *3) Remove a name*  *q) quit*  *3*  *What name do you want to remove?Ethan*  *list.remove(x): x not in list*  *press Enter to continue* | |

In the example below some more helpful changes have been made. Rather than just printing the error, a message saying the name isn’t in the list appears. The other change is the except block only catches ValueErrors. This is because it would be silly for the code to output ‘your name is not in the list’ if there is a different type of error or something else was failing.

| *…*  *3) Remove a name*  *…*  ***elif*** *choice == '3':*  *name\_to\_remove =* ***input(****'What name do you want to remove?'****)***  ***try****:*  *names.remove(name\_to\_remove)*  ***except*** *ValueError: # If 'ValueError' occurs then run the except block.*  ***print(****'The name "{}" was not in the list.'.****format(****name\_to\_remove****))***  ***input(****'press Enter to continue'****)*** | |
| --- | --- |
| Shell Window:  *['Tony', 'Betsy', 'Peter']*  *What do you want to do?*  *1) Add a name*  *2) Select a random name*  *3) Remove a name*  *q) quit*  *3*  *What name do you want to remove?Ethan*  *The name "ethan" was not in the list.*  *press Enter to continue* | |

## 