Python Scripting

Who is this for ?

- The name kind of mentions it:
  - System administrators who wish to learn basic and advance python scripting.
- But it also can be useful for:
  - Junior DevOps who wish to gain knowledge of python programming.
  - Junior software developers who have no knowledge in regards version control.

Prerequisites

- *Nix based working computer: Unix, FreeBSD, Well known Linux Distribution.
- Basic knowledge of hardware.
- Basic understanding of CLI: Bash, or alike(Any POSIX should be fine), in some cases PowerShell.
- Minimal understanding of filesystem.
- Some **Programming** or **Scripting** experience with other programming/scripting languages can be useful but **not** necessary .

Course Topics

- Basic
- Scripting Intro
- Conditions
- Storing Data
- Scripting Loops
- Functions
- Object Oriented Programming

Course Topics (cont.)

- Exception Handling
- Working With Files
- Modules and Libraries
- Modules: system
- Modules: parameters
- Modules: network
- Modules: database

About Me

- Over 12 years of IT industry Experience.
- Fell in love with AS-400 unix system at IDF.
- 5 times tried to finish degree in computer science field
  - Between each semester, I tried to take IT course at various places.
    - Yes, one of them was A+.
    - Yes, one of them was Cisco.
    - Yes, one of them was RedHat course.
    - Yes, one of them was LPIC1 and Shell scripting.
    - No, others i learned alone.
    - No, not maintaining debian packages any more.

About Me (cont.)

- Over 7 years of sysadmin:
  - Shell scripting fanatic
  - Python developer
  - JS admirer
  - Golang fallen
  - Rust fan
- 5 years of working with devops
  - Git supporter
  - Vagrant enthusiast
  - Ansible consultant
  - Container believer
  - K8s user

You can find me on linked-in: Alex M. Schapelle

And Now

- Back to course material

Python : Scripting Initials

Before Scripting Initials

Hardware -> CPU/RAM/HDD

Before we dive into python or scripting, some basics needs to provided: For That we need to know about hardware components, their role in big picture and how we utilize it all:

- CPU: Cental Processing Unit
- RAM: Random Access Memory
- HDD/SSD: Hard Drive Disk/Solid State Drive

Before Scripting Initials

RAM -> Data Management -> Programming

Most of programs in RAM. When we run a program, it allocates memory in RAM (clears space for itself to execute), and starts to implement the lines of code provided by developer(programmer). In order to process compute logical or mathematical information, it schedules access to CPU. After computing data it goes to HDD/SSD in order to save permanent data that it has computed.

Before Scripting Initials

Operational System(OS)

OS, is only program that has access to everything on our hardware. Today’s software is mainly based on OS to get access to CPU,RAM and HDD. Although there are computer systems that are not build in the same way, and logic to access CPU,RAM and HDD/SSD can be different. for example:

- Routers
- Switches
- SoC
- RTos
- and so on.

Before Scripting Initials

Software Development

Software development is the process of conceiving, specifying, designing, programming, documenting, testing, and bug fixing involved in creating and maintaining applications, frameworks, or other software components. Software development involves writing and maintaining the source code, but in a broader sense, it includes all processes from the conception of the desired software through to the final manifestation of the software, typically in a planned and structured process

Before Scripting Initials

Compile and RunTime

- A compile time (or compile-time) describes the time window during which a computer program is compiled. The term is used as an adjective to describe concepts related to the context of program compilation, as opposed to concepts related to the context of program execution (runtime)

- A runtime system or runtime environment is a sub-system that exists both in the computer where a program is created, as well as in the computers where the program is intended to be run. The name comes from the compile time and runtime division from compiled languages, which similarly distinguishes the computer processes involved in the creation of a program (compilation) and its execution in the target machine (the run time).

Before Scripting Initials

Assembly, C, C++ or C#

The difference between compile time and run time is an example of what pointy-headed theorists call the phase distinction. It is one of the hardest concepts to learn, especially for people without much background in programming languages. To approach this problem, I find it helpful to ask

- What environment does the program satisfy?
- What can go wrong in this phase?
- If the phase succeeds, what are the post conditions (what do we know)?
- What are the inputs and outputs, if any?

Before Scripting Initials

Compile Time Environment

The program need not satisfy any environment. In fact, it needn’t be a well-formed program at all. You could feed this HTML to the compiler and watch it spill it out with errors…

- What can go wrong at compile time:
  - Syntax errors
  - Typechecking errors
  - (Rarely) compiler crashes

Before Scripting Initials

Compile Time Environment

- If the compiler succeeds, what do we know?
  - The program was well formed—a meaningful program in whatever language.
  - It’s possible to start running the program. (The program might fail immediately, but at least we can try.)
- What are the inputs and outputs?
  - Input was the program being compiled, plus any header files, interfaces, libraries, or other voodoo that it needed to import in order to get compiled.
  - Output is hopefully assembly code or relocatable object code or even an executable program. Or if something goes wrong, output is a bunch of error messages.

Before Scripting Initials

REPL - Read Evaluate Print Loop

A read–eval–print loop (REPL), also termed an interactive toplevel or language shell, is a simple interactive computer programming environment that takes single user inputs, executes them, and returns the result to the user; a program written in a REPL environment is executed piecewise

Before Scripting Initials

JavaScript, Ruby, Python and many more

We know nothing about the program’s environments:

- They are whatever the programmer put in. Run-time environment are rarely enforced by the compiler alone; it needs help from the programmer.
- What can go wrong are run-time errors:
  - Division by zero
  - Dereferencing a null pointer
  - Running out of memory

Before Scripting Initials

JavaScript, Ruby, Python and many more

- Also there can be errors that are detected by the program itself:
  - Trying to open a file that isn’t there
  - Trying find a web page and discovering that an alleged URL is not well formed
- If run-time succeeds, the program finishes (or keeps going) without crashing.
  - Inputs and outputs are entirely up to the programmer. Files, windows on the screen, network packets, jobs sent to the printer, you name it. If the program launches missiles, that’s an output, and it happens only at run time :-)

Before Scripting Initials

Keywords and Identifiers

Keywords in Python

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	is

Python Script Intro

Namespaces

`Namespaces` are the constructs used for organizing the names assigned to the objects in a program. Essentially, A namespace is a collection of names and the details of the objects referenced by the names.

What is a built-in namespace in Python?

A built-in namespace contains the names of built-in functions and objects. It is created while starting the python interpreter, exists as long as the interpreter runs, and is destroyed when we close the interpreter. It contains the names of built-in data types, exceptions and functions like `print()` and `input()`.

What is a global namespace in python?

Global namespaces are defined at the program or module level. It contains the names of objects defined in a module or the main program. A global namespace is created when the program starts and exists until the program is terminated by the python interpreter

Variables

- How to keep data on your memory? Identifier names
  - Case sensitive: my_var is not my_VAR
  - Should start with _ or a-z or A-Z
  - Can not be saved words

Variables

- Conventions:
- `_my_var`: mostly used for `internal` or `private` use, will not be imported in case of use as library or module
- `__my_vat`: used for OOP, mostly useful with `attributes`, in other cases usable, but not suggested
- `__my_vax__`: usable but also a syntax used by python for python_system and might create troubles, so do not use

Variables

Other conventions

- Variables: lower-case, words separated with underscore(snake_case)
- Constants: ALL UPPER-CASE, words separated with underscore(snake_case)
- Functions: lower-case, words separated with underscore(snake_case)
- Classes: CapWords, also known as camel-case
- Modules: lower-case, short names, can have underscore, but not required
- Packages: lower-case, short names, preferred not to have underscore

Statements and Comments

How to remember what you have done.... a year ago?

Essentially we need to write code that is readable. Some time additional explanation for rational is required, thus comments are the best place to start with.

Practice

- Create python script file where you:
- Define variable for your name with value of your name
- Define variable of your nickname with value of your nickname
- Define variable of your pet,(children can be considered as pets) with value of your pets name
- Save the file and run it.

Data types

- Its all about definitions, but cpu still needs to know what you mean.
  - Numbers: integers and floats (complex too)
  - Strings: combined characters
  - Boolean: True/False or 1/0
  - None: null/void/nil or anything that can describe ___
  - There are also data structures that enable us to store more than one value, but we'll cover those later:
    - Lists, Sets, Dictionaries and Tuples

Practice

- Create python script file where you: - create variable for holding your name - create variable for holding your age - create variable for holding your pets name - use `print()` function to print them all.
- Save the file and run it.

Data types

In python every data type has its own features, and these features are unique to each and every one of the types **only**. We'll demo most of the features but keep in mind that one must always address documentation to see updates and new features in order to learn.

- The link to python documentation: `https://docs.python.org/3.7`
- In case of absence of internet connection, use `help()` function to get description of specific command in python shell
- `[!]` Note: to each new version of python, there might be changes in already existing features yet new features will be added to newer version of python.

Data types

Integers

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length. - `int()` function enables us to cast data into integer

Data types

Floats

Float, or “floating point number” is a number, positive or negative, containing one or more decimals. - `float()` function enables us to cast data into floating point number - Float can also be scientific numbers with an “e” to indicate the power of 10.

Data types

Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks. ‘hello’ is the same as “hello”. You can display a string literal with the `print()`

- `str()` function enables us to cast data into strings > `iface=str(123) # will be saved as string`
- Assigning a string to a variable is done with the variable name followed by an equal sign and the string
- You can assign a multiline string to a variable by using three double quotes Or three single quotes
- Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

Data types

Strings (cont.)

- However, Python does not have a character data type, a single character is simply a string with a length of 1.
- To get the length of a string, use the `len()` function.
- To check if a certain phrase or character is present in a string, we can use the keyword `in`.
- To check if a certain phrase or character is NOT present in a string, we can use the keyword `not in`.

Data types

Boolean

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

- bool() function enables us to cast data into boolean > `bool("abc") # will evaluate as True`
- Almost any value is evaluated to True if it has some sort of content.
- Any string is True, except empty strings.
- Any number is True, except 0.
- Any list, tuple, set, and dictionary are True, except empty ones.

There are not many values that evaluate to False, except empty values, such as (), [], {}, “”, the number 0, and the value None. And of course the value False evaluates to False.

Data types

None

Representation of the absence of a value, often called null in other languages

Type conversion

- You can change anything, if you convert it correctly. (mostly)
  - int/float -> string
  - string -> int/float
  - boolean -> int/string
  - int -> boolean
  - string -> boolean
- rational: in future lessons, we’ll be using code provided by others and we’ll try to incorporate data that code provides in our use cases, so learning how to convert one data type to other on the go is something we should know.

Practice

- Create python script file where you: - create variable for holding your spouse (ife/husband) name - check and print the type of that variable. - create variable for holding your spouses age - convert age variable to type string - create variable to hold your credit card details (if you wish, you can send them over the email to me: Joking....) - `print()` them away.
- Save the file and run it.

Operators

- Good! You have types and where to keep em'... but what else we can do with them ?
- We use operators to work with data(data can be considered anything, but mostly variable and data-structures):
  - Math Operators: + - * / ** // %
  - Assignment Operators: = += -= *= /= %=
  - Compare Operators: <> <= => == !=
  - Logical Operators: and or not
  - Membership Operators: is is not
  - Bitwise Operators: & | ^ ~ >> <<

Operators Precedence

Python has it too, so here is table that describes precedence from highest to lowest

Operator	Description
(expressions...), [expressions...], {key: value...},	Binding or parenthesized expression, list display, dictionary display, set display
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
await x	Await expression
**	Exponentiation
+x, -x, ~x	Positive, negative, bitwise NOT
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+, -	Addition and subtraction
<<, >>	Shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
if – else	Conditional expression
lambda	Lambda expression
:=	Assignment expression

Practice

- Create python script file where you:
  - create variable for wifi/eth/network that you are connected to, and put in it that name.
  - create variable for your laptops hostname and save the name in it.
  - concatenate the variable above with new variable named `my_con`.
  - `print()` the variable
  - add id (or any number) of your connection to my_con variable and re-print it.

I/O and Import

- Working with OS StdIn and StdOut or StdErr
- Working with files and standard library of python
    - We’ll cover this later on.

Slices

- You can return a range of characters by using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string.
- By leaving out the **start** index, the range will start at the first character
- By leaving out the **end** index, the range will go to the end
- Use negative indexes to start the slice from the end of the string

Practice

- Open REPL and create variable `iface` with value of your default interface
  - print value of the variable
  - present 3rd to 7th strings
  - show 2nd character from the end
  - check of 1st letter is `capital()`

String Format

Python has a set of built-in methods that you can use on strings.

- upper()
- lower()
- strip() : rstrip() lstrip()
- replace()
- split()
- format()
- is.. : islower() isupper() isdigit()
- find()
- index()

String concatenation

To concatenate, or combine, two strings you can use the + operator.

String escape characters

Code	Result
'	Single Quote
\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

Practice

- Create a python script
  - request from user to input some data **twice**
    - on the first input add back slash as part of input
      - print the input immediately
    - on the second input add double quote as part input
      - seek for double quote and print its position

Summary

Python Script Flow Control

Python Script Flow Control

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

Python Script Flow Control

Python Conditions and If statements (cont.)

These conditions can be used in several ways, most commonly in “if statements” and loops. An “if statement” is written by using the `if` keyword. The condition the `if` keyword is checking, is called `expression`. Expression specifies the conditions which are based on `Boolean` expression. When a Boolean expression is evaluated it produces either a value of true or false. If the expression evaluates true the same amount of indented statement(s) following if will be executed. This group of the statement(s) is called a block.

Python Script Flow Control

Indentation

```
var=5
     var=5 # this will be error
```

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Python Script Flow Control

If .. else

`if .. else` statement in python has two blocks: one following the `if` expression and other following the `else` clause. if the expression evaluates to `true` the same amount of indented statements(s) following if will be executed and if the expression evaluates to false the same amount of indented statements(s) following else will be executed

Python Script Flow Control

else

Python evaluates each expression (i.e. the condition) one by one and if a true condition is found the statement(s) block under that expression will be executed. If no true condition is found the statement(s) block under else will be executed.

The `else` keyword catches anything which isn’t caught by the preceding conditions.

Python Script Flow Control

Nested if .. else statement

In general nested if-else statement is used when we want to check more than one conditions, under existing condition. Conditions are executed from top to bottom and check each condition whether it evaluates to true or not. If a true condition is found the statement(s) block associated with the condition executes otherwise it goes to next condition.

```
if true:
    if true:
        print('True')
```

Define a negative if

If a condition is true the `not` operator is used to reverse the logical state, then logical `not` operator will make it false.

```
if not false:
    if true:
        print('True')
```

Practice

- Run in REPL and write the same code in code editor.(try to incorporate all that we have learned)
  - Print “Hello World” if a is greater than b.
  - Print “Hello World” if a is not equal to b.
  - Print “Yes” if a is equal to b, otherwise print “No”.
  - Print “1” if a is equal to b, print “2” if a is greater than b, otherwise print “3”.
  - Print “Hello” if a is equal to b, and c is equal to d.
  - Print “Hello” if a is equal to b, or if c is equal to d.
  - What is wrong with this: `sh if 5 > 2: print("Five is greater than two!")`

Python Script Arrays

Arrays as an idea

An array is structure: a form in which we can keep information for later use. That aside, array has very specific description: `An array is a collection of same type of elements which are sheltered under a common name.` for `c` programming language array:

```
  int array1={1,2,3,4,5}
```

Yet, when it comes to Python language, there are no `Arrays` per-se. Instead we'll be working with other ways, also known as data-structures, to store information. Although there is array library added in python3, we will `not` be working with it.

Python Script Arrays

Lists

Lists are used to store multiple items in a single variable. Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage. Lists are created using square brackets:

```
thislist = ["tomato", "cabbage", "cucumber"]
print(thislist)
```

Python Script Arrays

List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

Python Script Arrays

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

`[!]` Note: There are some list methods that will change the order, but in general: the order of the items will not change.

Python Script Arrays

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Python Script Arrays

Allow Duplicates

Since lists are indexed, lists can have items with the same value: Lists allow duplicate values:

```
thislist = ["tomato", "cabbage", "cucumber", "tomato", "lettuce"]
print(thislist)
```

Python Script Arrays

List Length

To determine how many items a list has, use the len() function:

Print the number of items in the list:

```
 thislist = ["tomato", "cabbage", "cucumber"]
print(len(thislist))
```

List Items - Data Types

List items can be of any data type: String, int and boolean data types:

```
list1 = ["tomato", "cabbage", "cucumber"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types: A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"]
```

Python Script Arrays

`type()` Function

From Python’s perspective, lists are defined as objects with the data type ‘list’:

What is the data type of a list?

```
mylist = ["tomato", "cabbage", "cucumber"]
print(type(mylist))
```

Python Script Arrays

The `list()` Constructor

It is also possible to use the list() constructor when creating a new list.

Using the list() constructor to make a List:

```
thislist = list(("tomato", "cabbage", "cucumber")) # note the double round-brackets
print(thislist)
```

Python Script Arrays

Access Items

List items are indexed and you can access them by referring to the index number:

```
 Print the second item of the list:
thislist = ["tomato", "cabbage", "cucumber"]
print(thislist[1])
```

`[!]`Note: The first item has index 0.

Python Script Arrays

Negative Indexing

Negative indexing means start from the end

-1 refers to the last item, -2 refers to the second last item etc.

```
 Print the last item of the list:
thislist = ["tomato", "cabbage", "cucumber"]
print(thislist[-1])
```

Python Script Arrays

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items. Return the third, fourth, and fifth item:

```
thislist = ["tomato", "cabbage", "cucumber", "mushroom", "lettuce", "celery", "carrot"]
print(thislist[2:5])
```

`[!]`Note: The search will start at index 2 (included) and end at index 5 (not included). `[!]`Note: Remember that the first item has index 0.

Python Script Arrays

Range of Indexes (cont.)

By leaving out the start value, the range will start at the first item: This returns the items from the beginning to, but NOT including, “lettuce”:

```
thislist = ["tomato", "cabbage", "cucumber", "mushroom", "lettuce", "celery", "carrot"]
print(thislist[:4])
```

Python Script Arrays

Range of Indexes (cont.)

By leaving out the end value, the range will go on to the end of the list: This returns the items from “cucumber” to the end:

```
thislist = ["tomato", "cabbage", "cucumber", "mushroom", "lettuce", "celery", "carrot"]
print(thislist[2:])
```

Python Script Arrays

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the list: This returns the items from “mushroom” (-4) to, but NOT including “carrot” (-1):

```
thislist = ["tomato", "cabbage", "cucumber", "mushroom", "lettuce", "celery", "carrot"]
print(thislist[-4:-1])
```

Python Script Arrays

Check if Item Exists

To determine if a specified item is present in a list use the in keyword: Check if “tomato” is present in the list:

```
thislist = ["tomato", "cabbage", "cucumber"]
if "tomato" in thislist:
  print("Yes, 'tomato' is in the fruits list")
```

Python Script Arrays

Change Item Value

To change the value of a specific item, refer to the index number:

Change the second item:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist[1] = "onion"
print(thislist)
```

Python Script Arrays

Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

Change the values “cabbage” and “lettuce” with the values “onion” and “nori”:

```
thislist = ["tomato", "cabbage", "cucumber", "mushroom", "lettuce", "celery", "carrot"]
thislist[1:3] = ["onion", "nori"]
print(thislist)
```

Python Script Arrays

Change a Range of Item Values (cont.)

If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

Change the second value by replacing it with two new values:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist[1:2] = ["onion", "nori"]
print(thislist)
```

`[!]` Note: The length of the list will change when the number of items inserted does not match the number of items replaced.

If you insert less items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

Python Script Arrays

Change a Range of Item Values (cont.)

Change the second and third value by replacing it with one value:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist[1:3] = ["nori"]
print(thislist)
```

Python Script Arrays

Insert Items

To insert a new list item, without replacing any of the existing values, we can use the insert() method.

The `insert()` method inserts an item at the specified index:

Insert “nori” as the third item:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist.insert(2, "nori")
print(thislist)
```

`[!]`Note: As a result of the above, the list will now contain 4 items.

Append Items

To add an item to the end of the list, use the `append()` method:

Using the append() method to append an item:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist.append("lemongrass")
print(thislist)
```

Insert Items

To insert a list item at a specified index, use the `insert()` method.

The insert() method inserts an item at the specified index:

Insert an item as the second position:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist.insert(1, "lemongrassgrass")
print(thislist)
```

`[!]` Note: As a result of the s above, the lists will now contain 4 items.

Extend List

To append elements from another list to the current list, use the `extend()` method.

Add the elements of tropical to thislist:

```
thislist = ["tomato", "cabbage", "cucumber"]
otherlist = ["wasabi", "zucchini", "papaya"]
thislist.extend(otherlist)
print(thislist)
```

The elements will be added to the end of the list.

Add Any Iterable

The extend() method does not have to append lists, you can add any iterable object (tuples, sets, dictionaries etc.).

Add elements of a tuple to a list:

```
thislist = ["tomato", "cabbage", "cucumber"]
thistuple = ("potato", "lemongrass")
thislist.extend(thistuple)
print(thislist)
```

Remove Specified Item

The remove() method removes the specified item.

Remove “cabbage”:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist.remove("cabbage")
print(thislist)
```

Remove Specified Index

The pop() method removes the specified index.

Remove the second item:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist.pop(1)
print(thislist)
```

If you do not specify the index, the pop() method removes the last item.

Remove the last item:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist.pop()
print(thislist)
```

The del keyword

also removes the specified index:

Remove the first item:

```
thislist = ["tomato", "cabbage", "cucumber"]
del thislist[0]
print(thislist)
```

The del keyword

can also delete the list completely.

Delete the entire list:

```
thislist = ["tomato", "cabbage", "cucumber"]
del thislist
```

Clear the List

The clear() method empties the list.

The list still remains, but it has no content.

Clear the list content:

```
thislist = ["tomato", "cabbage", "cucumber"]
thislist.clear()
print(thislist)
```

Sort List Alphanumerically

List objects have a sort() method that will sort the list alphanumerically, ascending, by default:

Sort the list alphabetically:

```
thislist = ["lemongrass", "wasabi", "potato", "zucchini", "cabbage"]
thislist.sort()
print(thislist)
```

Sort the list numerically:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

Sort Descending

To sort descending, use the keyword argument reverse = True:

Sort the list descending:

```
thislist = ["lemongrass", "wasabi", "potato", "zucchini", "cabbage"]
thislist.sort(reverse = True)
print(thislist)
```

Sort the list descending:

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```

Case Insensitive Sort

By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters:

Case sensitive sorting can give an unexpected result:

```
thislist = ["cabbage", "lemongrass", "potato", "lettuce"]
thislist.sort()
print(thislist)
```

Luckily we can use built-in functions as key functions when sorting a list.

So if you want a case-insensitive sort function, use str.lower as a key function:

Perform a case-insensitive sort of the list:

```
thislist = ["cabbage", "lemongrass", "potato", "lettuce"]
thislist.sort(key = str.lower)
print(thislist)
```

Reverse Order

What if you want to reverse the order of a list, regardless of the alphabet?

The reverse() method reverses the current sorting order of the elements.

Reverse the order of the list items:

```
thislist = ["cabbage", "lemongrass", "potato", "lettuce"]
thislist.reverse()
print(thislist)
```

Copy a List

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method copy().

Make a copy of a list with the copy() method:

```
thislist = ["tomato", "cabbage", "cucumber"]
mylist = thislist.copy()
print(mylist)
```

Copy a List

Another way to make a copy is to use the built-in method list().

Make a copy of a list with the list() method:

```
 thislist = ["tomato", "cabbage", "cucumber"]
mylist = list(thislist)
print(mylist)
```

Tuples

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

Create a Tuple:

```
thistuple = ("tomato", "cabbage", "cucumber")
print(thistuple)
```

Practice

- create python script:
  - has variable meal with type of list
  - require from user to insert his lunch parts, with loop.
  - insert lunch parts to meal list
  - insert main course of lunch as separate list in to meal list
  - sort both list in reverse order
  - append to meal list dessert you wish you had.
  - print the meal list in loop

Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index [0], the second item has index [1] etc. Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change. Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created. Allow Duplicates

Since tuples are indexed, they can have items with the same value:

Tuples allow duplicate values:

```
thistuple = ("tomato", "cabbage", "cucumber", "tomato", "lettuce")
print(thistuple)
```

Tuple Length

To determine how many items a tuple has, use the len() function:

Print the number of items in the tuple:

```
 thistuple = ("tomato", "cabbage", "cucumber")
print(len(thistuple))
```

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

One item tuple, remember the comma:

```
thistuple = ("tomato",)
print(type(thistuple))
```

NOT a tuple

```
 thistuple = ("tomato")
print(type(thistuple))
```

Tuple Items - Data Types

Tuple items can be of any data type:

String, int and boolean data types:

```
tuple1 = ("tomato", "cabbage", "cucumber")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

A tuple can contain different data types:

A tuple with strings, integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

type()

From Python’s perspective, tuples are defined as objects with the data type ‘tuple’:

What is the data type of a tuple?

```
mytuple = ("tomato", "cabbage", "cucumber")
print(type(mytuple))
```

The tuple() Constructor

It is also possible to use the tuple() constructor to make a tuple.

Using the tuple() method to make a tuple:

```
thistuple = tuple(("tomato", "cabbage", "cucumber")) # note the double round-brackets
print(thistuple)
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Print the second item in the tuple:

```
thistuple = ("tomato", "cabbage", "cucumber")
print(thistuple[1])
```

`[!]` Note: The first item has index 0.

Negative Indexing

Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

Print the last item of the tuple:

```
 thistuple = ("tomato", "cabbage", "cucumber")
print(thistuple[-1])
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

Return the third, fourth, and fifth item:

```
 thistuple = ("tomato", "cabbage", "cucumber", "lemongrass", "potato", "soybean", "wasabi")
print(thistuple[2:5])
```

`[!]` Note: The search will start at index 2 (included) and end at index 5 (not included).

`[!]` Note: Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

This returns the items from the beginning to, but NOT included, “potato”:

```
 thistuple = ("tomato", "cabbage", "cucumber", "lemongrass", "potato", "soybean", "wasabi")
print(thistuple[:4])
```

By leaving out the end value, the range will go on to the end of the list:

This returns the items from “lettuce” and to the end:

```
 thistuple = ("tomato", "cabbage", "cucumber", "lemongrass", "potato", "soybean", "wasabi")
print(thistuple[2:])
```

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

This returns the items from index -4 (included) to index -1 (excluded)

```
 thistuple = ("tomato", "cabbage", "cucumber", "lemongrass", "potato", "soybean", "wasabi")
print(thistuple[-4:-1])
```

Check if Item Exists

To determine if a specified item is present in a tuple use the in keyword:

Check if “tomato” is present in the tuple:

```
thistuple = ("tomato", "cabbage", "cucumber")
if "tomato" in thistuple:
  print("Yes, 'tomato' is in the fruits tuple")
```

Change Tuple Values

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Convert the tuple into a list to be able to change it:

```
x = ("tomato", "cabbage", "cucumber")
y = list(x)
y[1] = "potato"
x = tuple(y)

print(x)
```

Add Items

Since tuples are immutable, they do not have a built-in `append()` method, but there are other ways to add items to a tuple.

1. Convert into a list: Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Convert the tuple into a list, add “lemongrass”, and convert it back into a tuple:

```
thistuple = ("tomato", "cabbage", "cucumber")
y = list(thistuple)
y.append("lemongrass")
thistuple = tuple(y)
```

1. Add tuple to a tuple. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Create a new tuple with the value “lemongrass”, and add that tuple:

```
thistuple = ("tomato", "cabbage", "cucumber")
y = ("lemongrass",)
thistuple += y

print(thistuple)
```

`[!]` Note: When creating a tuple with only one item, remember to include a comma after the item, otherwise it will not be identified as a tuple.

Remove Items

`[!]`Note: You cannot remove items in a tuple.

Tuples are unchangeable, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Convert the tuple into a list, remove “tomato”, and convert it back into a tuple:

```
thistuple = ("tomato", "cabbage", "cucumber")
y = list(thistuple)
y.remove("tomato")
thistuple = tuple(y)
```

Or you can delete the tuple completely:

The del keyword can delete the tuple completely:

```
thistuple = ("tomato", "cabbage", "cucumber")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Unpacking a Tuple

When we create a tuple, we normally assign values to it. This is called “packing” a tuple:

Packing a tuple:

```
fruits = ("tomato", "cabbage", "cucumber")
```

But, in Python, we are also allowed to extract the values back into variables. This is called “unpacking”:

Unpacking a tuple:

```
fruits = ("tomato", "cabbage", "cucumber")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

`[!]` Note: The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

Using Asterisk*

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list:

Assign the rest of the values as a list called “red”:

```
fruits = ("tomato", "cabbage", "cucumber", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

Add a list of values the “tropic” variable:

```
fruits = ("tomato", "wasabi", "papaya", "zucchini", "lettuce")

(green, *tropic, red) = fruits

print(green)
print(tropic)
print(red)
```

Join Two Tuples

To join two or more tuples you can use the + operator:

Join two tuples:

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the * operator:

Multiply the fruits tuple by 2:

```
 fruits = ("tomato", "cabbage", "cucumber")
mytuple = fruits * 2

print(mytuple)
```

Sets

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is unordered, unchangeable*, and unindexed.

`[!]` Note: Set items are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.

Create a Set:

```
thisset = {"tomato", "cabbage", "cucumber"}
print(thisset)
```

`[!]` Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Set Items

Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

Once a set is created, you cannot change its items, but you can remove items and add new items.

Duplicates Not Allowed

Sets cannot have two items with the same value.

Duplicate values will be ignored:

```
thisset = {"tomato", "cabbage", "cucumber", "tomato"}

print(thisset)
```

len()

Get the Length of a Set

To determine how many items a set has, use the len() method.

Get the number of items in a set:

```
thisset = {"tomato", "cabbage", "cucumber"}

print(len(thisset))
```

Set Items - Data Types

Set items can be of any data type:

String, int and boolean data types:

```
set1 = {"tomato", "cabbage", "cucumber"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

A set can contain different data types:

A set with strings, integers and boolean values:

```
set1 = {"abc", 34, True, 40, "male"}
```

type()

From Python’s perspective, sets are defined as objects with the data type ‘set’:

What is the data type of a set?

```
myset = {"tomato", "cabbage", "cucumber"}
print(type(myset))
```

The set() Constructor

It is also possible to use the set() constructor to make a set.

Using the set() constructor to make a set:

```
thisset = set(("tomato", "cabbage", "cucumber")) # note the double round-brackets
print(thisset)
```

Access Items

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the `in` keyword.

Loop through the set, and print the values:

```
thisset = {"tomato", "cabbage", "cucumber"}

for x in thisset:
  print(x)


Check if "cabbage" is present in the set:
thisset = {"tomato", "cabbage", "cucumber"}

print("cabbage" in thisset)
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

To add items from another set into the current set, use the update() method.

Add elements from tropical into thisset:

```
thisset = {"tomato", "cabbage", "cucumber"}
tropical = {"zucchini", "wasabi", "papaya"}

thisset.update(tropical)

print(thisset)
```

Add Any Iterable

The object in the update() method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

Add elements of a list to at set:

```
thisset = {"tomato", "cabbage", "cucumber"}
mylist = ["potato", "lemongrass"]

thisset.update(mylist)

print(thisset)
```

Remove Item

To remove an item in a set, use the remove(), or the discard() method.

Remove “cabbage” by using the remove() method:

```
thisset = {"tomato", "cabbage", "cucumber"}

thisset.remove("cabbage")

print(thisset)
```

`[!]` Note: If the item to remove does not exist, remove() will raise an error.

Remove “cabbage” by using the discard() method:

```
thisset = {"tomato", "cabbage", "cucumber"}
thisset.discard("cabbage")
print(thisset)
```

`[!]` Note: If the item to remove does not exist, discard() will NOT raise an error.

Remove Item

You can also use the pop() method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the pop() method is the removed item.

Remove the last item by using the pop() method:

```
thisset = {"tomato", "cabbage", "cucumber"}
x = thisset.pop()
print(x)
print(thisset)
```

`[!]` Note: Sets are unordered, so when using the pop() method, you do not know which item that gets removed.

Remove Item

The clear() method empties the set:

```
thisset = {"tomato", "cabbage", "cucumber"}

thisset.clear()

print(thisset)
```

Remove Item

The del keyword will delete the set completely:

```
thisset = {"tomato", "cabbage", "cucumber"}

del thisset

print(thisset)
```

Join Two Sets

There are several ways to join two or more sets in Python.

You can use the union() method that returns a new set containing all items from both sets, or the update() method that inserts all the items from one set into another:

The union() method returns a new set with all items from both sets:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

The update() method inserts the items in set2 into set1:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set1.update(set2)
print(set1)
```

`[!]` Note: Both union() and update() will exclude any duplicate items.

Keep ONLY the Duplicates

The intersection_update() method will keep only the items that are present in both sets.

Keep the items that exist in both set x, and set y:

```
x = {"tomato", "cabbage", "cucumber"}
y = {"google", "microsoft", "tomato"}

x.intersection_update(y)

print(x)
```

The intersection() method will return a new set, that only contains the items that are present in both sets.

Return a set that contains the items that exist in both set x, and set y:

```
x = {"tomato", "cabbage", "cucumber"}
y = {"google", "microsoft", "tomato"}

z = x.intersection(y)

print(z)
```

Keep All, But NOT the Duplicates

The symmetric_difference_update() method will keep only the elements that are NOT present in both sets.

Keep the items that are not present in both sets:

```
x = {"tomato", "cabbage", "cucumber"}
y = {"google", "microsoft", "tomato"}

x.symmetric_difference_update(y)

print(x)
```

The symmetric_difference() method will return a new set, that contains only the elements that are NOT present in both sets.

Return a set that contains all items from both sets, except items that are present in both:

```
x = {"tomato", "cabbage", "cucumber"}
y = {"google", "microsoft", "tomato"}

z = x.symmetric_difference(y)

print(z)
```

Dictionary

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and does not allow duplicates.

As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Dictionaries are written with curly brackets, and have keys and values:

Create and print a dictionary:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Print the “brand” value of the dictionary:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

Duplicate values will overwrite existing values:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

Dictionary Length

To determine how many items a dictionary has, use the len() function:

Print the number of items in the dictionary:

```
print(len(thisdict))
```

Dictionary Items - Data Types

The values in dictionary items can be of any data type:

String, int, boolean, and list data types:

```
thisdict =  {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

type()

From Python’s perspective, dictionaries are defined as objects with the data type ‘dict’:

Print the data type of a dictionary:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(type(thisdict))
```

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Get the value of the “model” key:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
```

There is also a method called get() that will give you the same result:

Get the value of the “model” key:

```
x = thisdict.get("model")
```

Get Keys

The keys() method will return a list of all the keys in the dictionary.

Get a list of the keys:

```
x = thisdict.keys()
```

The list of the keys is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Add a new item to the original dictionary, and see that the keys list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

Get Values

The values() method will return a list of all the values in the dictionary.

Get a list of the values:

```
x = thisdict.values()
```

The list of the values is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

Make a change in the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Add a new item to the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["color"] = "red"
```

Get Items

The items() method will return each item in a dictionary, as tuples in a list.

Get a list of the key:value pairs

```
x = thisdict.items()
```

The returned list is a view of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

Make a change in the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Add a new item to the original dictionary, and see that the items list gets updated as well:

```
 car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword:

Check if “model” is present in the dictionary:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Change Values

You can change the value of a specific item by referring to its key name:

Change the “year” to 2018:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Update Dictionary

The update() method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

Update the “year” of the car by using the update() method:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Update Dictionary

The update() method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

Add a color item to the dictionary by using the update() method:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"color": "red"})
```

Removing Items

There are several methods to remove items from a dictionary:

The pop() method removes the item with the specified key name:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

The del keyword removes the item with the specified key name:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

The del keyword can also delete the dictionary completely:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

The clear() method empties the dictionary:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Copy a Dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method copy().

Make a copy of a dictionary with the copy() method:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Another way to make a copy is to use the built-in function dict().

Make a copy of a dictionary with the dict() function:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

Create a dictionary that contain three dictionaries:

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

if you want to add three dictionaries into a new dictionary:

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

Python Script loops

For Loop

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string). This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages. With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
veggies = ["tomato", "cabbage", "cucumber"]
for x in veggies:
  print(x)
```

For Loop (cont.)

Even strings are iterable objects, they contain a sequence of characters:

```
for x in "cucumber":
  print(x)
```

For Loop (cont.)

With the `break` statement we can stop the loop before it has looped through all the items:

```
veggies = ["tomato", "cabbage", "cucumber"]
for x in veggies:
  print(x)
  if x == "cucumber":
    break
```

For Loop (cont.)

With the continue statement we can stop the current iteration of the loop, and continue with the next:

```
veggies = ["tomato", "cabbage", "cucumber"]
for x in veggies:
  if x == "veggies":
    continue
  print(x)
```

For Loop (cont.)

Range Function

To loop through a set of code a specified number of times, we can use the range() function, The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):
  print(x)
```

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

```
for x in range(2, 6):
  print(x)
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3):

```
 for x in range(2, 30, 3):
  print(x)
```

For Loop (cont.)

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

`[!]` Note: The else block will NOT be executed if the loop is stopped by a break statement. Break the loop when x is 3, and see what happens with the else block:

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
```

For Loop (cont.)

A nested loop is a loop inside a loop. The “inner loop” will be executed one time for each iteration of the “outer loop”:

```
adj = ["red", "yellow", "green"]
veggies = ["tomato", "cabbage", "cucumber"]

for x in adj:
  for y in veggies:
    print(x, y)
```

For Loop (cont.)

Pass Statement

for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

```
for x in [0, 1, 2]:
  pass
```

Loop Through a List

You can loop through the list items by using a for loop:

Print all items in the list, one by one:

```
thislist = ["tomato", "cabbage", "cucumber"]
for x in thislist:
  print(x)
```

Learn more about for loops in our Python For Loops Chapter.

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the range() and len() functions to create a suitable iterable.

Print all items by referring to their index number:

```
thislist = ["tomato", "cabbage", "cucumber"]
for i in range(len(thislist)):
  print(thislist[i])
```

The iterable created in the above is [0, 1, 2].

Practice

- Import `subprocess` library
- Use `getoutput` function to get output of your systems ip interface information
- Loop through output and get interfaces that are up

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

A short hand for loop that will print all items in a list:

```
thislist = ["tomato", "cabbage", "cucumber"]
[print(x) for x in thislist]
```

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Based on a list of veggies, you want a new list, containing only the veggies with the letter “a” in the name.

Without list comprehension you will have to write a for statement with a conditional test inside:

```
veggies = ["tomato", "cabbage", "cucumber", "soybean", "wasabi"]
newlist = []

for x in veggies:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

With list comprehension you can do all that with only one line of code:

```
veggies = ["tomato", "cabbage", "cucumber", "soybean", "wasabi"]

newlist = [x for x in veggies if "a" in x]

print(newlist)
```

The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

Condition

The condition is like a filter that only accepts the items that valuate to True.

Only accept items that are not “tomato”:

```
newlist = [x for x in veggies if x != "tomato"]
```

The condition `if x != "tomato"` will return True for all elements other than “tomato”, making the new list contain all veggies except “tomato”.

The condition is optional and can be omitted:

With no if statement:

```
newlist = [x for x in veggies]
Iterable
if x != "tomato"
```

Expression

The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Set the values in the new list to upper case:

```
newlist = [x.upper() for x in veggies]
```

You can set the outcome to whatever you like: Set all values in the new list to ‘hello’:

```
newlist = ['hello' for x in veggies]
```

The expression can also contain conditions, not like a filter, but as a way to manipulate the outcome:

Return “lemongrass” instead of “cabbage”:

```
newlist = [x if x != "cabbage" else "lemongrass" for x in veggies]
```

The expression in the above says:

“Return the item if it is not cabbage, if it is cabbage return lemongrass”.

Loop Through a Tuple

You can loop through the tuple items by using a for loop.

Iterate through the items and print the values:

```
thistuple = ("tomato", "cabbage", "cucumber")
for x in thistuple:
  print(x)
```

Loop Through the Index Numbers

You can also loop through the tuple items by referring to their index number. Use the range() and len() functions to create a suitable iterable.

Print all items by referring to their index number:

```
thistuple = ("tomato", "cabbage", "cucumber")
for i in range(len(thistuple)):
  print(thistuple[i])
```

Practice

- Import `ipaddress` library
- Create variable that uses `ip_address` method with network segment
- Loop through the items in the variables `host()` data.

While Loop

With the while loop we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 8:
  print(i)
  i += 1
```

`[!]` Note: remember to increment i, or else the loop will continue forever.

The while loop requires relevant variables to be ready, in this we need to define an indexing variable, i, which we set to 1.

While Loop (cont.)

With the break statement we can stop the loop even if the while condition is true: Exit the loop when i is 3:

```
i = 1
while i < 8:
  print(i)
  if i == 3:
    break
  i += 1
```

While Loop (cont.)

With the continue statement we can stop the current iteration, and continue with the next:

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

While Loop (cont.)

With the else statement we can run a block of code once when the condition no longer is true: Print a message once the condition is false:

```
i = 1
while i < 8:
  print(i)
  i += 1
else:
  print("i is no longer less than 8")
```

Using a While Loop with Lists

You can loop through the list items by using a while loop. Use the len() function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes. Remember to increase the index by 1 after each iteration.

Print all items, using a while loop to go through all the index numbers

```
thislist = ["tomato", "cabbage", "cucumber"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```

Using a While Loop with Tuples

You can loop through the list items by using a while loop. Use the len() function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by referring to their indexes. Remember to increase the index by 1 after each iteration.

Print all items, using a while loop to go through all the index numbers:

```
thistuple = ("tomato", "cabbage", "cucumber")
i = 0
while i < len(thistuple):
  print(thistuple[i])
  i = i + 1
```

Practice

- Import `sys` library
- request from user provide your with parameters with `argv()` function/method.
- while the amount of elements will not be 10, continue requesting from user to provide parameters
- count the amount of parameters provided and let the user know how many more are left

Python Script Function

Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can `return` data as a result.

Creating a Function

In Python a function is defined using the def keyword:

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

```
def my_function():
  print("Hello from a function")

my_function()
```

Practice

- Create python function in REPL:
  - Imports `datetime` library.
  - That prints current date by using `datetime.date.today()` function.
- Create python script that:
  - Has function named print_triangle
  - Prints triangle using string.

Arguments

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(laptop_model):
  print(" lenovo"+ laptop_model )

my_function("Thinkpad")
my_function("Ideapad")
my_function("Thinkcenter")
```

Arguments are often shortened to args in Python documentations.

Parameters or Arguments?

The terms parameter and argument can be used for the same thing: information that are passed into a function. From a function’s perspective: - A parameter is the variable listed inside the parentheses in the function definition. - An argument is the value that is sent to the function when it is called.

Practice

- Create function in REPL:
  - that takes argument of your name, converts it to capital letters, and prints it out.
- Create python script with function named `list_of_files`:
- function should take an argument of `path` to some folder
- is should import os library with `import os`
- use function named `os.listdir()` and should print string of files in provided in the `path`

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(laptop_brand, laptop_model):
  print(laptop_brand + " " + laptop_model)

my_function("lenovo", "Thinkpad")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

This function expects 2 arguments, but gets only 1:

```
def my_function(laptop_brand, laptop_model):
  print(laptop_brand + " " + laptop_model)

my_function("lenovo")
```

Practice

- Create python script that takes 2 arguments with `input()`:
- imports `os` library and create folder with first argument and subfolder with second argument.
- use `os._exists()` to validated before run that folders do not exists.
- if they do, do nothing.
- if they do not, create them.

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*laptops):
  print("The newest laptop is " + laptops[2])

my_function("Alien", "Thinkpad", "MSI")
```

Arbitrary Arguments are often shortened to `*args` in Python documentations.

Practice

- Create python script with function that:
  -

Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

```
def my_function(laptop2, laptops1, laptop0):
  print("The oldest laptop is " + laptop0)

my_function(laptop0 = "thinkpad_x220", laptop1 = "thinkpad_x250", laptop2 = "thinkpad_t470")
```

The phrase Keyword Arguments are often shortened to `kwargs` in Python documentations.

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: `**` before the parameter name in the function definition. This way the function will receive a dictionary of arguments, and can access the items accordingly. If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**laptop):
  print("His last model is " + laptop["model"])

my_function(brand = "lenovo", model = "thinkpad_T470")
```

Arbitrary Keyword Arguments are often shortened to `**kwargs` in Python documentations.

Default Parameter Value

The following shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

```
def my_function(country = "Israel"):
  print("I am from " + country)

my_function("Georgia")
my_function("Russia")
my_function()
my_function("Italy")
```

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
 # The upper() method prints the string in upper case but it is not returning:
def my_function(foods):
  for food in foods:
    print(x)

grill = ["shwarma", "kebab", "fillet_minion"]

my_function(grill)
```

Return Values

To let a function return a value, use the return statement:

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

The pass Statement

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

```
def myfunction():
  pass
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this , tri_recursion() is a function that we have defined to call itself (“recurse”). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Recursion

```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion  Results")
```

List Of Builtin Functions

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object

Builtin Functions(cont.)

callable() | Returns True if the specified object is callable, otherwise False chr() | Returns a character from the specified Unicode code. classmethod() | Converts a method into a class method compile() | Returns the specified source as an object, ready to be executed complex() | Returns a complex number delattr() | Deletes the specified attribute (property or method) from the specified object dict() | Returns a dictionary (Array)

Builtin Functions(cont.)

dir() | Returns a list of the specified object’s properties and methods divmod() | Returns the quotient and the remainder when argument1 is divided by argument2 enumerate() | Takes a collection (e.g. a tuple) and returns it as an enumerate object eval() | Evaluates and executes an expression exec() | Executes the specified code (or object) filter() | Use a filter function to exclude items in an iterable object float() | Returns a floating point number format() | Formats a specified value frozenset() | Returns a frozenset object

Builtin Functions(cont.)

getattr() | Returns the value of the specified attribute (property or method) globals() | Returns the current global symbol table as a dictionary hasattr() | Returns True if the specified object has the specified attribute (property/method) hash() | Returns the hash value of a specified object help() | Executes the built-in help system hex() | Converts a number into a hexadecimal value id() | Returns the id of an object input() | Allowing user input int() | Returns an integer number

Builtin Functions(cont.)

isinstance() | Returns True if a specified object is an instance of a specified object issubclass() | Returns True if a specified class is a subclass of a specified object iter() | Returns an iterator object len() | Returns the length of an object list() | Returns a list locals() | Returns an updated dictionary of the current local symbol table map() | Returns the specified iterator with the specified function applied to each item max() | Returns the largest item in an iterable memoryview() | Returns a memory view object

Builtin Functions(cont.)

min() | Returns the smallest item in an iterable next() | Returns the next item in an iterable object() | Returns a new object oct() | Converts a number into an octal open() | Opens a file and returns a file object ord() | Convert an integer representing the Unicode of the specified character pow() | Returns the value of x to the power of y print() | Prints to the standard output device

Builtin Functions(cont.)

property() | Gets, sets, deletes a property range() | Returns a sequence of numbers, starting from 0 and increments by 1 (by default) repr() | Returns a readable version of an object reversed() | Returns a reversed iterator round() | Rounds a numbers set() | Returns a new set object setattr() | Sets an attribute (property/method) of an object slice() | Returns a slice object sorted() | Returns a sorted list @staticmethod() | Converts a method into a static method

Builtin Functions(cont.)

str() | Returns a string object sum() | Sums the items of an iterator super() | Returns an object that represents the parent class tuple() | Returns a tuple type() | Returns the type of an object vars() | Returns the **dict** property of an object zip() | Returns an iterator, from two or more iterators

Python Object Oriented Programming

Object Oriented Programming

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let’s take an example:

A parrot is an object, as it has the following properties:

- name, age, color as attributes
- singing, dancing as behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don’t Repeat Yourself).

In Python, the concept of OOP follows some basic principles: Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, `obj` is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots. Example 1: Creating Class and Object in Python

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Output

```
Blu is a bird
Woo is also a bird
```

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object. Example 2 : Creating Methods in Python

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Output

```
Blu sings 'Happy'
Blu is now dancing
```

In the above program, we define two methods i.e `sing()` and `dance()`. These are called instance methods because they are called on an instance object i.e `blu`.

4 Pillars of OOP

- abstraction: hiding enformation

Python Objects and Classes

Python is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stresses on objects.

An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.

We can think of a class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

As many houses can be made from a house's blueprint, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called instantiation. Defining a Class in Python

Like function definitions begin with the def keyword in Python, class definitions begin with a class keyword.

The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.

Here is a simple class definition.

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores `__`. For example, `__doc__` gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class Person:
    "This is a person class"
    age = 10
```

Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

```
>>> harry = Person()
```

This will create a new object instance named harry. We can access the attributes of objects using the object name prefix.

Attributes may be data or method. Methods of an object are corresponding functions of that class.

This means to say, since Person.greet is a function object (attribute of class), Person.greet will be a method object.

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# create a new object of Person class
harry = Person()

# Output: <function Person.greet>
print(Person.greet)

# Output: <bound method Person.greet of <__main__.Person object>>
print(harry.greet)

# Calling object's greet() method
# Output: Hello
harry.greet()
```

Output

Inheritance in Python

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class. Python Inheritance Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code. Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides n and magnitude of each side as a list called sides.

The inputSides() method takes in the magnitude of each side and dispSides() displays these side lengths.

Method Overriding in Python

In the above example, notice that `__init__()` method was defined in both classes, Triangle as well Polygon. When this happens, the method in the derived class overrides that in the base class. This is to say, `__init__()` in Triangle gets preference over the `__init__` in Polygon.

Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling Polygon.`__init__()` from `__init__()` in Triangle).

A better option would be to use the built-in function `super()`. So, `super().__init__(3)` is equivalent to `Polygon.__init__(self,3)` and is preferred. To learn more about the `super()` function in Python, visit Python super() function with `help()` function in you python REPL.

Two built-in functions isinstance() and issubclass() are used to check inheritances.

The function isinstance() returns True if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class object.

```
>>> isinstance(t,Triangle)
True

>>> isinstance(t,Polygon)
True

>>> isinstance(t,int)
False

>>> isinstance(t,object)
True
```

Similarly, `issubclass()` is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False

>>> issubclass(Triangle,Polygon)
True
```

Python Multilevel Inheritance

We can also inherit from a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class.

An example with corresponding visualization is given below.

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```

Here, the Derived1 class is derived from the Base class, and the Derived2 class is derived from the Derived1 class.

Method Resolution Order in Python

Every class in Python is derived from the object class. It is the most base type in Python.

So technically, all other classes, either built-in or user-defined, are derived classes and all objects are instances of the object class.

```
 # Output: True
print(issubclass(list,object))

# Output: True
print(isinstance(5.5,object))

# Output: True
print(isinstance("Hello",object))
```

Multiple Inheritance Visualization

Visualizing Multiple Inheritance in Python

```
# Demonstration of MRO

class X:
    pass


class Y:
    pass


class Z:
    pass


class A(X, Y):
    pass


class B(Y, Z):
    pass


class M(B, A, Z):
    pass

# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
#  <class '__main__.A'>, <class '__main__.X'>,
#  <class '__main__.Y'>, <class '__main__.Z'>,
#  <class 'object'>]

print(M.mro())
```

Output

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.X'>, <class '__ma
```

Python Operator Overloading

You can change the meaning of an operator in Python depending upon the operands used. In this tutorial, you will learn how to use operator overloading in Python Object Oriented Programming.

Python Operator Overloading

Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

class Point: def **init**(self, x=0, y=0): self.x = x self.y = y

p1 = Point(1, 2) p2 = Point(2, 3) print(p1+p2)

Output

Traceback (most recent call last): File “”, line 9, in print(p1+p2) TypeError: unsupported operand type(s) for +: ‘Point’ and ‘Point’

Here, we can see that a TypeError was raised, since Python didn’t know how to add two Point objects together.

However, we can achieve this task in Python through operator overloading. But first, let’s get a notion about special functions. Python Special Functions

Class functions that begin with double underscore `__` are called special functions in Python.

These functions are not the typical functions that we define for a class. The `__init__()` function we defined above is one of them. It gets called every time we create a new object of that class.

There are numerous other special functions in Python. Visit Python Special Functions to learn more about them

Overloading Comparison Operators

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.

Suppose we wanted to implement the less than symbol < symbol in our Point class.

Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

```
# overloading the less than operator
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# use less than
print(p1<p2)
print(p2<p3)
print(p1<p3)
```

Output

```
True
False
False
```

Similarly, the special functions that we need to implement, to overload other comparison operators

Exception handling: Try.. Except

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message. These exceptions can be handled using the try statement:

The try block will generate an exception, because x is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed. Without the try block, the program will crash and raise an error: This statement will raise an error, because x is not defined:

```
print(x)
```

Practice

create python script that : - uses variable that has not been defined with value - uses `except` statement to evade exit and asks for value for that variable.

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Practice

create python script that : - import `sys` library - set variable to keep in value of argument. - uses variable that has not been defined with value - uses `except` statement to evade exit and asks for value for that variable. - uses another `except` statement in case the the value of variable is not convertible to int

Else

You can use the else keyword to define a block of code to be executed if no errors were raised:

In this , the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Practice

- create python script that has a function:
- import `ipaddress` library
- requires ipv4 without cidr from user.
- in case cidr is added, print error message.
- in case of ip address in not valid, print error

Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
except:
  print("Something went wrong when opening the file")
```

The program can continue, without leaving the file object open.

Practice

- create python script:
- import `Path` from `pathlib` library
- import `sys` library
- pass value from `sys.argv[1]` to variable
- use `exists()` to check evaluation whether path exists`.

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the raise keyword. Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

The `raise` keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Raise a TypeError if x is not an integer:

```
x = "ip-address"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

Practice : Error handling

- create python script that will list the content of current folder:
- in case there is no `.txt` extension files, raise error

File Handling

File handling is an important part of any programming language.

Python has several functions for creating, reading, updating, and deleting files.

In this module we’ll go over the capabilietes of Python to handle files in various ways, by reading, writing, parsing data saved in files.

File Handling

The key function for working with files in Python is the open() function.

The `open()` function takes two parameters; filename, and mode.

There are four different methods (modes) for opening a file:

- “r” | Read - Default value. Opens a file for reading, error if the file does not exist
- “r+”| Read and Write - Opens file and allows to write in to it
    - In case file does not exists, will get error
- “a” | Append - Opens a file for appending only, creates the file if it does not exist
- “a+”| Read and Append - Opens files for reading and apprending to the file.
- “w” | Write - Opens a file for writing:
    - In case file does **not exist**, it will be **created**
    - In case file **exists**, the data will be **squahsed**
- “w+”| Read and Write - Opens file and allows to write in to it
    - In case file does **not exist**, it will be **created**
    - In case file **exists**, the data will be **squahsed**
- “x” | Create - Creates the specified file, returns an error if the file exists

File Habdling Table

Mode	**r**	**r+**	**w**	**w+**	**a**	**a+**
read	+	+		+		+
Write		+	+	+	+	+
Write after seek		+	+	+		
Create			+	+	+	+
Truncate			+	+		
Position at start	+	+	+	+		
Position at end					+	+

In addition you can specify if the file should be handled as binary or text mode

- “t” - Text - Default value. Text mode
- “b” - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because “r” for read, and “t” for text are the default values, you do not need to specify them.

`[!]` Note: Make sure the file exists, or else you will get an error.

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

```
alex@vaiolabs:~$ cat demofile.txt
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in `open()` function.

The open() function returns a file object, which has a read() method for reading the content of the file:

```
f = open("sh_ip_int.txt", "r")
print(f.read())
```

If the file is located in a different location, you will have to specify the file path, like this:

Open a file on a different location:

```
 f = open("/home/aschapelle/Desktop/sh_ip_int.txt", "r")
print(f.read())
```

Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Return the 5 first characters of the file:

```
 f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the readline() method:

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

By calling readline() two times, you can read the two first lines:

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files

It is a good practice to always close the file when you are done with it.

Close the file when you are finish with it:

```
 f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

`[!]` Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

- “a” | Append - will append to the end of the file
- “w” | Write - will overwrite any existing content

Open the file “demofile2.txt” and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()
```

open and read the file after the appending:

```
f = open("demofile2.txt", "r")
print(f.read())
```

Open the file “demofile3.txt” and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

`[!]` Note: the “w” method will overwrite the entire file.

Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

- “x” | Create - will create a file, returns an error if the file exist
- “a” | Append - will create a file if the specified file does not exist
- “w” | Write - will create a file if the specified file does not exist

Create a file called “myfile.txt”:

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

Remove the file “demofile.txt”:

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Check if file exists, then delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the os.rmdir() method:

Remove the folder “myfolder”:

```
import os
os.rmdir("myfolder")
```

`[!]` Note: You can only remove empty folders.

Libs

Python uses some terms that you may not be familiar with if you’re coming from a different language. Among these are **scripts**, **modules**, **packages**, and **libraries**.

A **script** is a Python file that’s intended to be run directly. When you run it, it should do something. This means that scripts will often contain code written outside the scope of any classes or functions.

A **module** is a Python file that’s intended to be imported into scripts or other modules. It often defines members like classes, functions, and variables intended to be used in other files that import it.

Libs (cont.)

A **package** is a collection of related modules that work together to provide certain functionality. These modules are contained within a folder and can be imported just like any other modules. This folder will often contain a special **init** file that tells Python it’s a package, potentially containing more modules nested within subfolders

A **library** is an umbrella term that loosely means “a bundle of code.” These can have tens or even hundreds of individual modules that can provide a wide range of functionality. Matplotlib is a plotting library. The Python Standard Library contains hundreds of modules for performing common tasks, like sending emails or reading JSON data. What’s special about the Standard Library is that it comes bundled with your installation of Python, so you can use its modules without having to download them from anywhere.

These are not strict definitions. Many people feel these terms are somewhat open to interpretation. Script and module are terms that you may hear used interchangeably.

3rd Party Libraries

In the remaining part of this course, we’ll be working with various packages that are and also aren’t included with Python by default.Many programming languages offer a package manager that automates the process of installing, upgrading, and removing third-party packages. Python is no exception. The de facto package manager for Python is called pip. Historically, pip had to be downloaded and installed separately from Python. As of Python 3.4, it’s now included with most distributions of the language.

Working With Modules

A module is a file containing Python code that can be reused in other Python code files. Technically, every Python file that you've created while reading this book is a module, but you haven't seen how to use code from one module inside another. There are four main advantages to breaking a program into modules:

- **Simplicity**: Modules are focused on a single problem.
- **Maintainability**: Small files are better than large files.
- **Reusability**: Modules reduce duplicate code.
- **Scoping**: Modules have their own namespaces

Creating Modules
Importing One Module Into Another
Import Statement Variations
Working With Packages
Importing Modules From Packages
Import Statement Variations for Packages
Guidelines for Importing Packages
Importing Modules From Subpackages

Python Script Modules

MATH

These functions cannot be used with complex numbers; use the functions of the same name from the cmath module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don’t is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

RANDOM

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

OS

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see open(), if you want to manipulate paths, see the os.path module, and if you want to read all the lines in all the files on the command line see the fileinput module. For creating temporary files and directories see the tempfile module, and for high-level file and directory handling see the shutil module.

OS.PATH

This module implements some useful functions on pathnames. To read or write files see open(), and for accessing the filesystem see the os module. The path parameters can be passed as strings, or bytes, or any object implementing the os.PathLike protocol.

TIME

This module provides various time-related functions. For related functionality, see also the datetime and calendar modules.

Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

GETPASS

This module gave access to gaining portable password input, while also keeping the aspect of security, by not saving clear password and converting it to

GLOB

The glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but *, ?, and character ranges expressed with [] will be correctly matched. This is done by using the os.scandir() and fnmatch.fnmatch() functions in concert, and not by actually invoking a subshel

SHUTIL

The shutil module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the os module.

IO

SYS

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

SUBPROCESS

The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

os.system os.spawn*

Information about how the subprocess module can be used to replace these modules and functions can be found in the following sections

LOGGING

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules

Python Script Parameters

Output and Input

Positional parameters - sys module

sys.stdout.write

sys.stdin.read / readline / readlines

Argument parameters - argparse module

Python Script Network Modules

SOCKET

Sockets have a long history. Their use originated with ARPANET in 1971 and later became an API in the Berkeley Software Distribution (BSD) operating system released in 1983 called Berkeley sockets.

When the Internet took off in the 1990s with the World Wide Web, so did network programming. Web servers and browsers weren't the only applications taking advantage of newly connected networks and using sockets. Client-server applications of all types and sizes came into widespread use.

Today, although the underlying protocols used by the socket API have evolved over the years, and new ones have developed, the low-level API has remained the same.

The most common type of socket applications are client-server applications, where one side acts as the server and waits for connections from clients. This is the type of application that you'll be creating in this tutorial. More specifically, you'll focus on the socket API for Internet sockets, sometimes called Berkeley or BSD sockets. There are also Unix domain sockets, which can only be used to communicate between processes on the same host.

Overview

Python's socket module provides an interface to the Berkeley sockets API. This is the module that you'll use in this tutorial.

The primary socket API functions and methods in this module are:

- socket()
- .bind()
- .listen()
- .accept()
- .connect()
- .connect_ex()

JSON

YAML

XMLTODICT

PEXPECT

Telnetlib

PARAMIKO

NETMIKO

DataBases

SQLITE
MYSQL
MONGODB
SQLALCHEMY

Python Script Gui

TKINTER
PyGTK
wxPython# Python Programming With Networking Libraries
Programming With Open Source

In todays software development we tend to combine between our wish to develop fast and to develop secure. Fortunately there are a lot of programming languages and development ecosystems that provide that possibility. One of those languages is `Python`: `Open Source General Purpose Programming Language` with broad standard library and large community of developers and contributors.

Who Is This Course For ?

- Software developers who would like to learn python.
- Junior/senior sysadmins who have no knowledge programming and automation with python.
- DevOps engineers who would like to learn software development.
- SRE engineers who would like educate themselves with python.

Course Topics

- Scripting intro
- History.
- Types of programming languages.
- What is Python ?
- How Python works ?
- What is required for Python?

Lesson 1

- History
  - How Python came to be
  - Basic explainer of programming concepts
- Environment Setup
  - Python install/compile
  - REPL and how to use it
  - Development tools
  - Version control for beginners
- Python Syntax
  - Spaces not tabs
  - Naming conventions
  - Basic program structures
  - Entry point and how to use it
- Data Types
  - Integers
  - Floats
  - Strings
  - Boolean
  - Operators

Lesson 2

- Recap
- Looping
  - Serial loops
  - Conditional loops
  - Operators in Loops
  - Looping thought data
- Functions
  - What is function ?
  - Variables in function
    - private
    - global
  - Operators and return
  - Conditions and return
  - Loops and return
  - `Main` function
  - Argument pass to function
  - Dynamic argument passing
  - Key Value argument pass
  - Positional arguments

Lesson 3

- Recap
- Exceptions
  - Use cases for errors
  - handling errors
  - types of errors
- Working with files
- Modules
  - What is a module
  - How to `import` modules
  - Importing without execution
  - Module path
  - Installing modules
  - Virtual environments
  - Debugging code with python
- Standard library
  - os
  - sys
  - subprocess
  - getpass

Lesson 4

- Recap
- CSV
- Json
- Xml-to-dict
- Yaml
- Socket
- Http handling with requests
- Telnet connection with telnetlib
- Expecting communications with pexpect
- File transfer with ftplib
- Ssh connections with paramiko
- Summary

Lesson 5

- Recap
- Ssh connections with netmiko
- Web Services
  - Rest and SOAP
- Web application
  - Flask
- End Project
- Summary