

Python Essentials

Python 101

What is a programming language? Set of conventions for authoring computer programs What's a program? A set of instructions for how a computer should react to various inputs and outputs

History Lesson:

- Punch cards - [read it for history sake](#)

-
- Assembly language:
 - Processor specific --> not portable
 - Binary correspondence
 - Easier to read than punch cards or binary
 - more info [here](#)

-
- "Higher level" languages
 - Fortran, Cobol, C, C++, C#:
 - Easier to read and write
 - Slightly more portable
 - more infor [here](#)

-
- Scripting languages:
 - JS, Python, Ruby, Groovy, Bash:
 - more info [here](#)

Types of languages

- Compiled Languages:
 - C, C++, C#, Objective-C
 - [read about compiler](#)
- Scripting languages:
 - Python, Ruby, Javascript, Lua
 - [read about run time environment](#)

Programming Language Styles

- Procedural
- Object Oriented

Procedural Languages

Procedural programming is a programming paradigm, derived from structured programming, based on the concept of the procedure call. Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

- Procedures/Routines/Subroutines/Functions that include series of functions
- [link](#)

Object Oriented Languages

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods). A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self").

- Represent objects
- Define their properties
- Define ways to manipulate them
- Object inheritance
- [link](#)

REPL: Python Interpreter

As mentioned [before](#) python is interpreted language. The interpretation for this kind of projects is done in a manner of REPL.

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 (i.e., single expressions), evaluates (executes) them, and returns the result to the user.

In a REPL, the user enters one or more expressions (rather than an entire compilation unit) and the REPL evaluates them and displays the results. The name read-eval-print loop comes from the names of the Lisp primitive functions which implement this functionality:

The [read](#) function accepts an expression from the user, and parses it into a data structure in memory. For instance, the user may enter the s-expression `(+ 1 2 3)`, which is parsed into a linked list containing four data elements.

The [eval](#) function takes this internal data structure and evaluates it. In Lisp, evaluating an s-expression beginning with the name of a function means calling that function on the arguments that make up the rest of the expression. So the function `+` is called on the arguments `1 2 3`, providing (yielding) the result `6`.

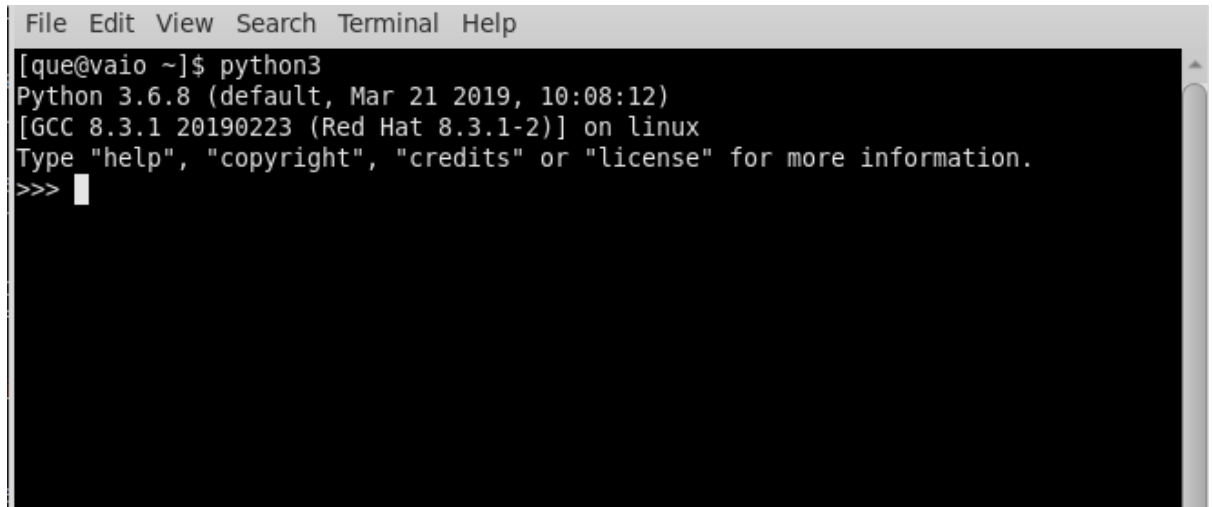
The [print](#) function takes the result provided (yielded) by `eval`, and prints it out to the user. If it is a complex expression, it may be pretty-printed to make it easier to understand.

The development environment then returns to the read state, creating a [loop](#), which terminates when the program is closed.

REPL's are interactive, by which i mean that they interact with the user by success, the command provided to REPL with work or by error, the command will [NOT](#) work. REPL usually looks same as command line, for

example:

- Open cmd/terminal of your OS
- Type in it `python3` command and you are in
- It should look like this:



```
File Edit View Search Terminal Help
[que@vaio ~]$ python3
Python 3.6.8 (default, Mar 21 2019, 10:08:12)
[GCC 8.3.1 20190223 (Red Hat 8.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```

- In case you see `python 2.7` instead of what you have in picture, then it means that you need to upgrade your current version of python to `python 3.6` or higher.

Saving the code

Although running in REPL is fun and all, one must always run the list of commands from somewhere, mostly because it misses point of programming if you'll run all the commands manually.

list of python (or any other high level language) commands that are saved in to the file is usually called as `code`.

To write code, we use a type of `text editor` to write and run it. There are many text editor on www, here is short list of ones that i'd suggest for this course:

- GEANY --> a lightweight IDE for general purpose of development.

```

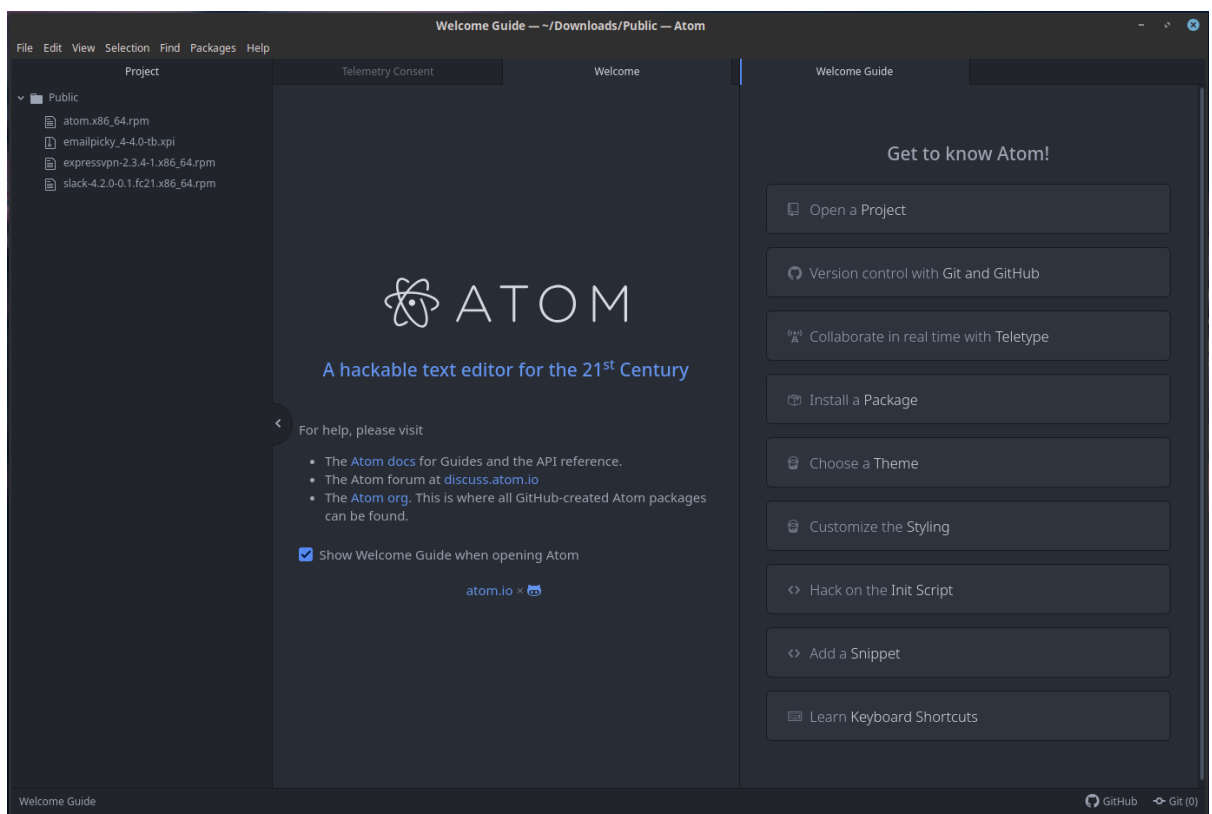
1 #!/usr/bin/env bash
2 # set -xe
3 #####
4 #Scripting name :
5 #Purpose       :
6 #Args          : adding
7 #Created by    : your nickname
8 #E-mail        : name@mail.com
9 #Version       : 1.0.0
10 #####
11
12 echo Hello, world!
13
14

```

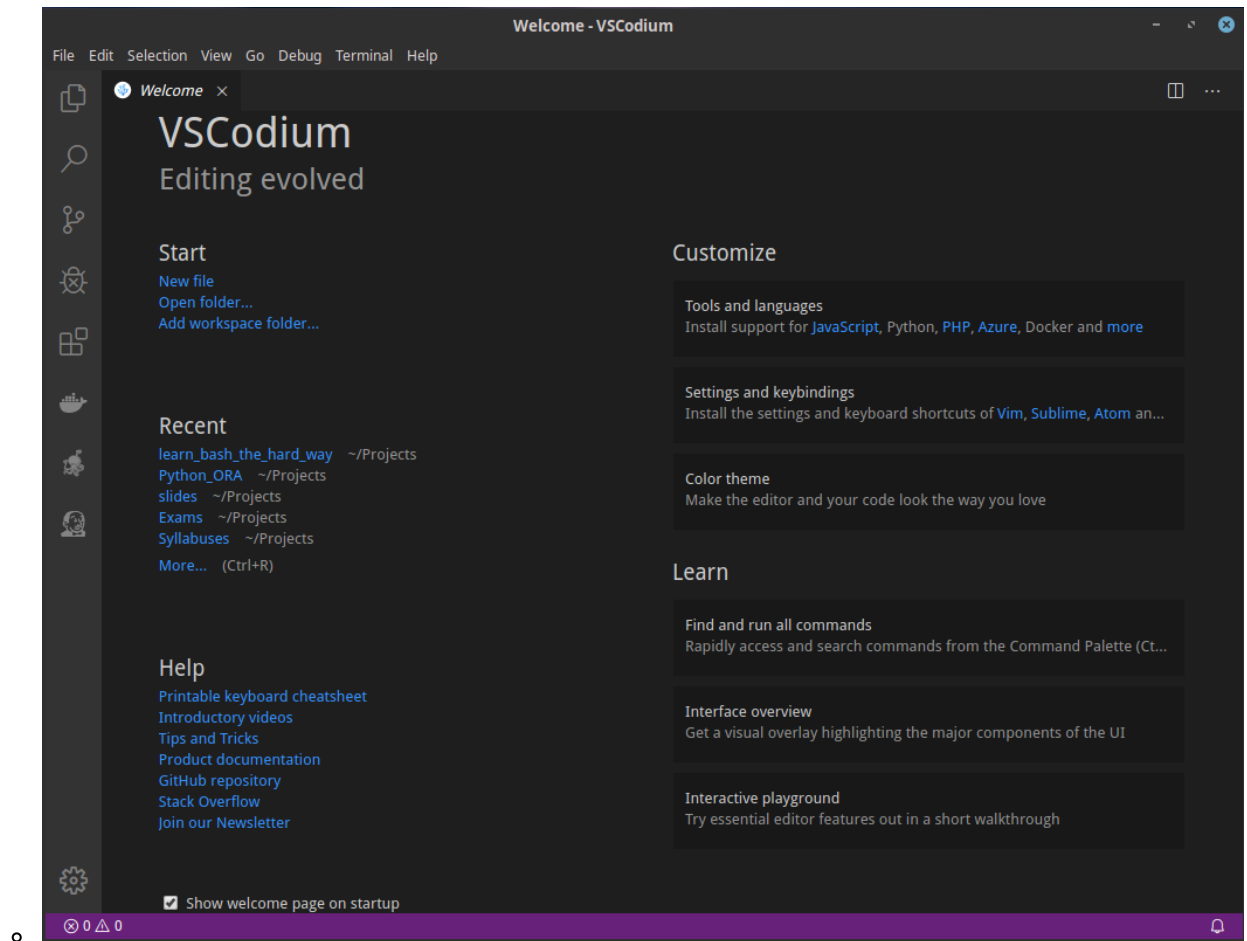
15:18:12: This is Geany 1.36.
 Status 15:18:12: Failed to load one or more session files.
 15:18:12: New file "untitled" opened.
 Compiler 15:18:30: File /home/que/tmp.sh saved.
 Messages

line: 1 / 14 col: 0 sel: 0 INS TAB MOD mode: LF encoding: UTF-8 filetype: Sh scope: unknown

- ATOM --> a extensible text editor with bunch tools that can be added.



- VSCODIUM --> a clone project of atom and vscode with more native OpenSource licensing and agenda.



Python Syntax

Python syntax can be executed by writing directly in the Command Line: [img]

Or by creating a python file on the server, using the .py file extension, and running it in the Command Line: [img]

Python Indentation

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code.

```
if 5 > 2:
    print("Five is greater than two!")    #<-- 4 space indentation
```

Python will give you an error if you skip the indentation:

```
if 5 > 2:
print("Five is greater than two!")    #<-- NO INDENTATION
```

Python Variables

In Python variables are created the moment you assign a value to it:

```
x = 5
y = "Hello, World!"
```

Python has no command for declaring a variable. Unlike other programming languages, Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.

Comments

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code. Creating a Comment

Comments starts with a #, and Python will ignore them:

```
#This is a comment
print("Hello, World!")
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

```
print("Hello, World!") #This is a comment
```

Comments does not have to be text to explain the code, it can also be used to prevent Python from executing code:

```
#print("Hello, World!")
print("Cheers, Mate!")
```

Multi Line Comments

Python does not really have a syntax for multi line comments.

To add a multiline comment you could insert a # for each line:

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Or, not quite as intended, you can use a multiline string. Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Python Data Types

Built-in Data Types

In programming, data type is an important concept. Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

Type	Value
Text	str
Numeric	int, float, complex
Sequence	list, tuple, range
Mapping	dict
Set	set, frozenset
Boolean	bool
Binary	bytes, bytearray

Getting the Data Type

You can get the data type of any object by using the `type()` function:

Print the data type of the variable `x`:

```
x = 5
print(type(x))
```

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

```
x = "Hello World" #      str
x = 20 #int
x = 20.5 #float
x = ["apple", "banana", "cherry"] #list
x = ("apple", "banana", "cherry") #tuple
x = range(6) #range
x = {"name" : "John", "age" : 36} #dict
x = {"apple", "banana", "cherry"} #set
x = frozenset({"apple", "banana", "cherry"}) #frozenset
x = True #bool
x = b"Hello" #bytes
x = bytearray(5) #bytearray
x = memoryview(bytes(5)) #memoryview
```

Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

```
x = str("Hello World") #      str
x = int(20) #int
x = float(20.5) #float
x = complex(1j) #complex
x = list(("apple", "banana", "cherry")) #list
x = tuple(("apple", "banana", "cherry")) #tuple
x = range(6) #range
x = dict(name="John", age=36) #dict
x = set(("apple", "banana", "cherry")) #set
x = frozenset(("apple", "banana", "cherry")) #frozenset
x = bool(5) #bool
x = bytes(5) #bytes
x = bytearray(5) #bytearray
x = memoryview(bytes(5)) #memoryview
```

Python Numbers

There are three numeric types in Python:

- int
- float

Variables of numeric types are created when you assign a value to them:

```
x = 1 # int
y = 2.8 # float
```


To verify the type of any object in Python, use the `type()` function:

```
print(type(x))  
print(type(y))
```

```
<!-- print(type(z)) -->
```

Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Integers:

```
x = 1  
y = 35656222554887711  
z = -3255522  
  
print(type(x))  
print(type(y))  
print(type(z))
```

Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Floats:

```
x = 1.10  
y = 1.0  
z = -35.59  
  
print(type(x))  
print(type(y))  
print(type(z))
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

Floats:

```
x = 35e3  
y = 12E4  
z = -87.7e100  
  
print(type(x))  
print(type(y))  
print(type(z))
```

Convert from one type to another:

```
x = 1 # int
y = 2.8 # float
```

convert from int to float: `a = float(x)`

convert from float to int: `b = int(y)`

```
print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

Note: You cannot convert complex numbers into another number type.

Python Strings

What is String in Python?

A string is a **sequence of characters**. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's. This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used. In Python, string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn more about Unicode from [here](#).

How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
# all of the following are equivalent
my_string = 'Hello'
print(my_string)

my_string = "Hello"
print(my_string)
```

```
my_string = '''Hello'''
print(my_string)

# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
               the world of Python"""
print(my_string)
```

When you run the program, the output will be:

```
Hello
Hello
Hello
Hello, welcome to
               the world of Python
```

How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an `IndexError`. The index must be an integer. We can't use float or other types, this will result into **TypeError**. Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator (colon).

```
str = 'programiz'
print('str = ', str)

#first character
print('str[0] = ', str[0])

#last character
print('str[-1] = ', str[-1])

#slicing 2nd to 5th character
print('str[1:5] = ', str[1:5])

#slicing 6th to 2nd last character
print('str[5:-2] = ', str[5:-2])
```

If we try to access index out of the range or use decimal number, we will get errors.

```
# index must be in range
>>> my_string[15]
...
IndexError: string index out of range
# index must be an integer
```

```
>>> my_string[1.5]
...
TypeError: string indices must be integers
```

Slicing can be best visualized by considering the index to be between the elements as shown below. If we want to access a range, we need the index that will slice the portion from the string.

How to change or delete a string?

Strings are immutable. This means that elements of a string cannot be changed once it has been assigned. We can simply reassign different strings to the same name.

```
>>> del my_string[1]
...
TypeError: 'str' object doesn't support item deletion
>>> del my_string
>>> my_string
...
NameError: name 'my_string' is not defined
```

Python String Operations

There are many operations that can be performed with string which makes it one of the most used datatypes in Python.

Concatenation of Two or More Strings

Joining of two or more strings into a single one is called concatenation.

The `+` operator does this in Python. Simply writing two string literals together also concatenates them.

The `*` operator can be used to repeat the string for a given number of times.

```
str1 = 'Hello'
str2 = 'World!'

# using +
print('str1 + str2 = ', str1 + str2)

# using *
print('str1 * 3 =', str1 * 3)
```

Writing two string literals together also concatenates them like `+` operator.

If we want to concatenate strings in different lines, we can use parentheses.

```
>>> # two string literals together
>>> 'Hello ' 'World!'
'Hello World!'
>>> # using parentheses
>>> s = ('Hello '
...      'World')
>>> s
'Hello World'
```

Iterating Through String

Using for loop we can iterate through a string. Here is an example to count the number of 'l' in a string.

```
count = 0
for letter in 'Hello World':
    if(letter == 'l'):
        count += 1
print(count, 'letters found')
```

String Membership Test

We can test if a sub string exists within a string or not, using the keyword in.

```
>>> 'a' in 'program'
True
>>> 'at' not in 'battle'
False
```

Built-in functions to Work with Python

Various built-in functions that work with sequence, works with string as well.

Some of the commonly used ones are enumerate() and len(). The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Similarly, len() returns the length (number of characters) of the string.

```
str = 'cold'

# enumerate()
list_enumerate = list(enumerate(str))
print('list(enumerate(str) = ', list_enumerate)

#character count
print('len(str) = ', len(str))
```

Python String Formatting

###Escape Sequence

If we want to print a text like -He said, "What's there?"- we can neither use single quote or double quotes. This will result into SyntaxError as the text itself contains both single and double quotes.

```
>>> print("He said, "What's there?")
...
SyntaxError: invalid syntax
>>> print('He said, "What's there?')
...
SyntaxError: invalid syntax
```

One way to get around this problem is to use triple quotes. Alternatively, we can use escape sequences.

An escape sequence starts with a backslash and is interpreted differently. If we use single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes. Here is how it can be done to represent the above text.

```
# using triple quotes
print('''He said, "What's there?''')

# escaping single quotes
print('He said, "What\'s there?')

# escaping double quotes
print("He said, \"What's there?\")
```

Here is a list of all the escape sequence supported by Python.

Escape Sequence in Python

Escape Sequence	Description
\newline	Backslash and newline ignored
\	Backslash
'	Single quote
"	Double quote
\a	ASCII Bell
\b	ASCII Backspace
\f	ASCII Formfeed

Escape Sequence	Description
\n	ASCII Linefeed
\r	ASCII Carriage Return
\t	ASCII Horizontal Tab
\v	ASCII Vertical Tab
\ooo	Character with octal value ooo
\xHH	Character with hexadecimal value HH

Here are some examples

```
>>> print("\\")
\  
>>> print("This is printed\nin two lines")
This is printed
in two lines
>>> print("This is \x48\x45\x58 representation")
This is HEX representation
```

Raw String to ignore escape sequence

Sometimes we may wish to ignore the escape sequences inside a string. To do this we can place r or R in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

```
>>> print("This is \x41 \ngood example")
This is A
good example
>>> print(r"This is \x41 \ngood example")
This is \x41 \ngood example
```

The format() Method for Formatting Strings

The format() method that is available with the string object is very versatile and powerful in formatting strings. Format strings contains curly braces {} as placeholders or replacement fields which gets replaced.

We can use positional arguments or keyword arguments to specify the order.

```
# default(implicit) order
default_order = "{}, {} and {}".format('John', 'Bill', 'Sean')
print('\n--- Default Order ---')
print(default_order)

# order using positional argument
positional_order = "{1}, {0} and {2}".format('John', 'Bill', 'Sean')
```

```
print('\n--- Positional Order ---')
print(positional_order)

# order using keyword argument
keyword_order = "{s}, {b} and {j}".format(j='John',b='Bill',s='Sean')
print('\n--- Keyword Order ---')
print(keyword_order)
```

The format() method can have optional format specifications. They are separated from field name using colon. For example, we can left-justify <, right-justify > or center ^ a string in the given space. We can also format integers as binary, hexadecimal etc. and floats can be rounded or displayed in the exponent format. There are a ton of formatting you can use. Visit [here](#) for all the string formatting available with the format() method.

```
>>> # formatting integers
>>> "Binary representation of {0} is {0:b}".format(12)
'Binary representation of 12 is 1100'
>>> # formatting floats
>>> "Exponent representation: {0:e}".format(1566.345)
'Exponent representation: 1.566345e+03'
>>> # round off
>>> "One third is: {0:.3f}".format(1/3)
'One third is: 0.333'
>>> # string alignment
>>> "|{:<10}|{: ^10}|{:>10}|".format('butter','bread','ham')
'|butter      | bread    |      ham|'
```

Old style formatting

We can even format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

```
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

Common Python String Methods

There are numerous methods available with the string object. The format() method that we mentioned above is one of them. Some of the commonly used methods are lower(), upper(), join(), split(), find(), replace() etc. Here is a complete list of all the built-in methods to work with strings in Python.

```
>>> "PrOgRaMiZ".lower()
'programiz'
```



```
>>> "PrOgRaMiZ".upper()
'PROGRAMIZ'
>>> "This will split all words into a list".split()
['This', 'will', 'split', 'all', 'words', 'into', 'a', 'list']
>>> ' '.join(['This', 'will', 'join', 'all', 'words', 'into', 'a',
'string'])
'This will join all words into a string'
>>> 'Happy New Year'.find('ew')
7
>>> 'Happy New Year'.replace('Happy', 'Brilliant')
'Brilliant New Year'
```

Python List

Python offers a range of compound datatypes often referred to as sequences. List is one of the most frequently used and very versatile datatype used in Python.

How to create a list?

In Python programming, a list is created by placing all the items (elements) inside a square bracket [], separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).

```
# empty list
my_list = []
# list of integers
my_list = [1, 2, 3]
# list with mixed datatypes
my_list = [1, "Hello", 3.4]
```

Also, a list can even have another list as an item. This is called nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

How to access elements from a list?

There are various ways in which we can access the elements of a list.

List Index

We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4. Trying to access an element other than this will raise an IndexError. The index

must be an integer. We can't use float or other types, this will result into `TypeError`. Nested list are accessed using nested indexing.

```
my_list = ['p', 'r', 'o', 'b', 'e']
# Output: p
print(my_list[0])
# Output: o
print(my_list[2])
# Output: e
print(my_list[4])
# Error! Only integer can be used for indexing
# my_list[4.0]
# Nested List
n_list = ["Happy", [2, 0, 1, 5]]
# Nested indexing
# Output: a
print(n_list[0][1])
# Output: 5
print(n_list[1][3])
```

Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_list = ['p', 'r', 'o', 'b', 'e']
# Output: e
print(my_list[-1])
# Output: p
print(my_list[-5])
```

How to slice lists in Python?

We can access a range of items in a list by using the slicing operator (colon).

```
my_list = ['p', 'y', 't', 'h', 'o', 'n', ' ', 'c', 'o', 'u', 'r', 's', 'e']
# elements 3rd to 5th
print(my_list[2:5])
# elements beginning to 4th
print(my_list[:-5])
# elements 6th to end
print(my_list[5:])
# elements beginning to end
print(my_list[:])
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two indices that will slice that portion from the list.

How to change or add elements to a list?

List are mutable, meaning, their elements can be changed unlike string or tuple.

We can use assignment operator (=) to change an item or a range of items.

```
# mistake values
odd = [2, 4, 6, 8]
# change the 1st item
odd[0] = 1
# Output: [1, 4, 6, 8]
print(odd)
# change 2nd to 4th items
odd[1:4] = [3, 5, 7]
# Output: [1, 3, 5, 7]
print(odd)
```

We can add one item to a list using append() method or add several items using extend() method.

```
odd = [1, 3, 5]
odd.append(7)
# Output: [1, 3, 5, 7]
print(odd)
odd.extend([9, 11, 13])
# Output: [1, 3, 5, 7, 9, 11, 13]
print(odd)
```

We can also use + operator to combine two lists. This is also called concatenation. The * operator repeats a list for the given number of times.

```
odd = [1, 3, 5]
# Output: [1, 3, 5, 9, 7, 5]
print(odd + [9, 7, 5])
#Output: ["re", "re", "re"]
print(["re"] * 3)
```

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

```
odd = [1, 9]
odd.insert(1, 3)
# Output: [1, 3, 9]
```

```
print(odd)
odd[2:2] = [5, 7]
# Output: [1, 3, 5, 7, 9]
print(odd)
```

How to delete or remove elements from a list?

We can delete one or more items from a list using the keyword `del`. It can even delete the list entirely.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
# delete one item
del my_list[2]
# Output: ['p', 'r', 'b', 'l', 'e', 'm']
print(my_list)
# delete multiple items
del my_list[1:5]
# Output: ['p', 'm']
print(my_list)
# delete entire list
del my_list
# Error: List not defined
print(my_list)
```

We can use `remove()` method to remove the given item or `pop()` method to remove an item at the given index. The `pop()` method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure). We can also use the `clear()` method to empty a list.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
my_list.remove('p')
# Output: ['r', 'o', 'b', 'l', 'e', 'm']
print(my_list)
# Output: 'o'
print(my_list.pop(1))
# Output: ['r', 'b', 'l', 'e', 'm']
print(my_list)
# Output: 'm'
print(my_list.pop())
# Output: ['r', 'b', 'l', 'e']
print(my_list)
my_list.clear()
# Output: []
print(my_list)
```

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
>>> my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
>>> my_list[2:3] = []
```

```
>>> my_list
['p', 'r', 'b', 'l', 'e', 'm']
>>> my_list[2:5] = []
>>> my_list
['p', 'r', 'm']
```

List Comprehension: Elegant way to create new List

List comprehension is an elegant and concise way to create new list from an existing list in Python. List comprehension consists of an expression followed by for statement inside square brackets. Here is an example to make a list with each item being increasing power of 2.

```
pow2 = [2 ** x for x in range(10)]
# Output: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
print(pow2)
```

This code is equivalent to

```
pow2 = []
for x in range(10):
    pow2.append(2 ** x)
```

A list comprehension can optionally contain more for or if statements. An optional if statement can filter out items for the new list. Here are some examples.

```
>>> pow2 = [2 ** x for x in range(10) if x > 5]
>>> pow2
[64, 128, 256, 512]
>>> odd = [x for x in range(20) if x % 2 == 1]
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> [x+y for x in ['Python ', 'C '] for y in ['Language', 'Programming']]
['Python Language', 'Python Programming', 'C Language', 'C
Programming']
```

Other List Operations in Python

List Membership Test

We can test if an item exists in a list or not, using the keyword in.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
# Output: True
```

```
print('p' in my_list)
# Output: False
print('a' in my_list)
# Output: True
print('c' not in my_list)
```

Iterating Through a List

Using a for loop we can iterate through each item in a list.

```
for fruit in ['apple', 'banana', 'mango']:
    print("I like", fruit)
```

Python Tuple

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas, in a list, elements can be changed.

Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Empty tuple
my_tuple = ()
print(my_tuple) # Output: ()

# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple) # Output: (1, 2, 3)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple) # Output: (1, "Hello", 3.4)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# Output: ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

A tuple can also be created without using parentheses. This is known as tuple packing.

```
my_tuple = 3, 4.6, "dog"
print(my_tuple)    # Output: 3, 4.6, "dog"

# tuple unpacking is also possible
a, b, c = my_tuple

print(a)           # 3
print(b)           # 4.6
print(c)           # dog
```

Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

```
my_tuple = ("hello")
print(type(my_tuple)) # <class 'str'>

# Creating a tuple having one element
my_tuple = ("hello",)
print(type(my_tuple)) # <class 'tuple'>

# Parentheses is optional
my_tuple = "hello",
print(type(my_tuple)) # <class 'tuple'>
```

Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

1. Indexing

We can use the index operator `[]` to access an item in a tuple where the index starts from 0.

So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an element outside of tuple (for example, 6, 7,...) will raise an `IndexError`.

The index must be an integer; so we cannot use float or other types. This will result in `TypeError`.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

```
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])    # 'p'
print(my_tuple[5])    # 't'

# IndexError: list index out of range
# print(my_tuple[6])
```

```
# Index must be an integer
# TypeError: list indices must be integers, not float
# my_tuple[2.0]

# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
print(n_tuple[0][3])      # 's'
print(n_tuple[1][1])      # 4
```

2. Negative Indexing

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_tuple = ('p','e','r','m','i','t')

# Output: 't'
print(my_tuple[-1])

# Output: 'p'
print(my_tuple[-6])
```

3. Slicing

We can access a range of items in a tuple by using the slicing operator - colon ":".

```
my_tuple = ('p','y','t','h','o','n',' ','c','o','u','r','s','e')

# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r')
print(my_tuple[:-7])

# elements 8th to end
# Output: ('i', 'z')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p','y','t','h','o','n',' ','c','o','u','r','s','e')
print(my_tuple[:])
```


Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.

Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed.

We can also assign a tuple to different values (reassignment).

```
my_tuple = (4, 2, 3, [6, 5])

# TypeError: 'tuple' object does not support item assignment
# my_tuple[1] = 9

# However, item of mutable element can be changed
my_tuple[3][0] = 9      # Output: (4, 2, 3, [9, 5])
print(my_tuple)

# Tuples can be reassigned
my_tuple = ('p','y','t','h','o','n',' ','c','o','u','r','s','e')

# Output: ('p','y','t','h','o','n',' ','c','o','u','r','s','e')
print(my_tuple)
```

We can use + operator to combine two tuples. This is also called concatenation.

We can also repeat the elements in a tuple for a given number of times using the * operator.

Both + and * operations result in a new tuple.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.

But deleting a tuple entirely is possible using the keyword del.

```
my_tuple = ('p','y','t','h','o','n',' ','c','o','u','r','s','e')

# can't delete items
# TypeError: 'tuple' object doesn't support item deletion
# del my_tuple[3]

# Can delete an entire tuple
del my_tuple

# NameError: name 'my_tuple' is not defined
print(my_tuple)
```

Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Python Tuple Method

Method	Description
count(x)	Returns the number of items x
index(x)	Returns the index of the first item that is equal to x

Some examples of Python tuple methods:

```
my_tuple = ('a','p','p','l','e',)

print(my_tuple.count('p')) # Output: 2
print(my_tuple.index('l')) # Output: 3
```

Other Tuple Operations

1. Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

```
my_tuple = ('a','p','p','l','e',)

# In operation
# Output: True
print('a' in my_tuple)

# Output: False
print('b' in my_tuple)

# Not in operation
```

```
# Output: True
print('g' not in my_tuple)
```

2. Iterating Through a Tuple

Using a for loop we can iterate through each item in a tuple.

```
# Output:
# Hello John
# Hello Kate
for name in ('John', 'Kate'):
    print("Hello", name)
```

Advantages of Tuple over List

Since tuples are quite similar to lists, both of them are used in similar situations as well.

However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuples are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Python Sets

What is a set in Python?

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). However, the set itself is mutable. We can add or remove items from it. Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

How to create a set?

A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function set(). It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.

```
# set of integers
my_set = {1, 2, 3}
```

```
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

Try the following examples as well.

```
# set do not have duplicates
# Output: {1, 2, 3, 4}
my_set = {1, 2, 3, 4, 3, 2}
print(my_set)

# set cannot have mutable items
# here [3, 4] is a mutable list
# If you uncomment line #12,
# this will cause an error.
# TypeError: unhashable type: 'list'

#my_set = {1, 2, [3, 4]}

# we can make set from a list
# Output: {1, 2, 3}
my_set = set([1, 2, 3, 2])
print(my_set)
```

Creating an empty set is a bit tricky.

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the set() function without any argument.

```
# initialize a with {}
a = {}

# check data type of a
# Output: <class 'dict'>
print(type(a))

# initialize a with set()
a = set()

# check data type of a
# Output: <class 'set'>
print(type(a))
```

How to change a set in Python?

Sets are mutable. But since they are unordered, indexing have no meaning. We cannot access or change an element of set using indexing or slicing. Set does not support it. We can add single element using the `add()` method and multiple elements using the `update()` method. The `update()` method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
# initialize my_set
my_set = {1, 3}
print(my_set)

# if you uncomment line 9,
# you will get an error
# TypeError: 'set' object does not support indexing

#my_set[0]

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2, 3, 4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4, 5], {1, 6, 8})
print(my_set)
```

How to remove elements from a set?

A particular item can be removed from set using methods, `discard()` and `remove()`.

The only difference between the two is that, while using `discard()` if the item does not exist in the set, it remains unchanged. But `remove()` will raise an error in such condition.

The following example will illustrate this.

```
# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

# remove an element
# Output: {1, 3, 5}
```

```
my_set.remove(6)
print(my_set)

# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)

# remove an element
# not present in my_set
# If you uncomment line 27,
# you will get an error.
# Output: KeyError: 2

#my_set.remove(2)
```

Similarly, we can remove and return an item using the `pop()` method.

Set being unordered, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all items from a set using `clear()`.

```
# initialize my_set
# Output: set of unique elements
my_set = set("HelloWorld")
print(my_set)

# pop an element
# Output: random element
print(my_set.pop())

# pop another element
# Output: random element
my_set.pop()
print(my_set)

# clear my_set
#Output: set()
my_set.clear()
print(my_set)
```

Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
```

Different Python Set Methods

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with set objects. Python Set Methods

Method	Description
<code>add()</code>	Adds an element to the set
<code>clear()</code>	Removes all elements from the set
<code>copy()</code>	Returns a copy of the set
<code>difference()</code>	Returns the difference of two or more sets as a new set
<code>difference_update()</code>	Removes all elements of another set from this set
<code>discard()</code>	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
<code>intersection()</code>	Returns the intersection of two sets as a new set
<code>intersection_update()</code>	Updates the set with the intersection of itself and another
<code>isdisjoint()</code>	Returns True if two sets have a null intersection
<code>issubset()</code>	Returns True if another set contains this set
<code>issuperset()</code>	Returns True if this set contains another set
<code>pop()</code>	Removes and returns an arbitrary set element. Raise <code>KeyError</code> if the set is empty
<code>remove()</code>	Removes an element from the set. If the element is not a member, raise a <code>KeyError</code>
<code>symmetric_difference()</code>	Returns the symmetric difference of two sets as a new set
<code>symmetric_difference_update()</code>	Updates a set with the symmetric difference of itself and another
<code>union()</code>	Returns the union of sets in a new set
<code>update()</code>	Updates the set with the union of itself and others

Other Set Operations

Set Membership Test

We can test if an item exists in a set or not, using the keyword `in`.

```
# initialize my_set
my_set = set("apple")

# check if 'a' is present
# Output: True
print('a' in my_set)

# check if 'p' is present
# Output: False
print('p' not in my_set)
```

Iterating Through a Set

Using a for loop, we can iterate through each item in a set.

```
>>> for letter in set("apple"):
...     print(letter)
...
a
p
e
l
```

Built-in Functions with Set

Built-in functions like `all()`, `any()`, `enumerate()`, `len()`, `max()`, `min()`, `sorted()`, `sum()` etc. are commonly used with set to perform different tasks. Built-in Functions with Set

Function	Description
<code>all()</code>	Return True if all elements of the set are true (or if the set is empty).
<code>any()</code>	Return True if any element of the set is true. If the set is empty, return False.
<code>enumerate()</code>	Return an enumerate object. It contains the index and value of all the items of set as a pair.
<code>len()</code>	Return the length (the number of items) in the set.
<code>max()</code>	Return the largest item in the set.
<code>min()</code>	Return the smallest item in the set.
<code>sorted()</code>	Return a new sorted list from elements in the set(does not sort the set itself).
<code>sum()</code>	Return the sum of all elements in the set.

Python Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets. Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the function `frozenset()`.

This datatype supports methods like `copy()`, `difference()`, `intersection()`, `isdisjoint()`, `issubset()`, `issuperset()`, `symmetric_difference()` and `union()`. Being immutable it does not have method that add or remove elements.

```
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])
```

Try these examples on Python shell.

```
>>> A.isdisjoint(B)
False
>>> A.difference(B)
frozenset({1, 2})
>>> A | B
frozenset({1, 2, 3, 4, 5, 6})
>>> A.add(3)
...
AttributeError: 'frozenset' object has no attribute 'add'
```

Python Dictionary

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a **key: value** pair.

Dictionaries are optimized to retrieve values when the key is known.

How to create a dictionary?

Creating a dictionary is as simple as placing items inside curly braces `{}` separated by comma. An item has a key and the corresponding value expressed as a pair, **key: value**. While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
# empty dictionary
my_dict = {}
# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}
# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}
```

```
# using dict()
my_dict = dict({1:'apple', 2:'ball'})
# from sequence having each item as a pair
my_dict = dict([(1, 'apple'), (2, 'ball')])
```

As you can see above, we can also create a dictionary using the built-in function dict().

How to access elements from a dictionary?

While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the get() method. The difference while using get() is that it returns None instead of KeyError, if the key is not found.

```
my_dict = {'name': 'Jack', 'age': 26}

# Output: Jack
print(my_dict['name'])

# Output: 26
print(my_dict.get('age'))

# Trying to access keys which doesn't exist throws error
# my_dict.get('address')
# my_dict['address']
```

How to change or add elements in a dictionary?

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator. If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

```
my_dict = {'name': 'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

How to delete or remove elements from a dictionary?

We can remove a particular item in a dictionary by using the method `pop()`. This method removes an item with the provided key and returns the value. The method, `popitem()` can be used to remove and return an arbitrary item (key, value) from the dictionary. All the items can be removed at once using the `clear()` method. We can also use the `del` keyword to remove individual items or the entire dictionary itself.

```
# create a dictionary
squares = {1:1, 2:4, 3:9, 4:16, 5:25}

# remove a particular item
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item
# Output: (1, 1)
print(squares.popitem())

# Output: {2: 4, 3: 9, 5: 25}
print(squares)

# delete a particular item
del squares[5]

# Output: {2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)

# delete the dictionary itself
del squares

# Throws Error
# print(squares)
```

Python Dictionary Methods

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples. Python Dictionary Methods

Method	Description
<code>clear()</code>	Remove all items from the dictionary.
<code>copy()</code>	Return a shallow copy of the dictionary.

Method	Description
<code>fromkeys(seq[, v])</code>	Return a new dictionary with keys from seq and value equal to v (defaults to None).
<code>get(key[,d])</code>	Return the value of key. If key doesnot exit, return d (defaults to None).
<code>items()</code>	Return a new view of the dictionary's items (key, value).
<code>keys()</code>	Return a new view of the dictionary's keys.
<code>pop(key[,d])</code>	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
<code>popitem()</code>	Remove and return an arbitrary item (key, value). Raises KeyError if the dictionary is empty.
<code>setdefault(key[,d])</code>	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
<code>update([other])</code>	Update the dictionary with the key/value pairs from other, overwriting existing keys.
<code>values()</code>	Return a new view of the dictionary's values

Here are a few example use of these methods.

```
marks = {}.fromkeys(['Math', 'English', 'Science'], 0)

# Output: {'English': 0, 'Math': 0, 'Science': 0}
print(marks)

for item in marks.items():
    print(item)

# Output: ['English', 'Math', 'Science']
list(sorted(marks.keys()))
```

Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python. Dictionary comprehension consists of an expression pair (key: value) followed by for statement inside curly braces `{}`. Here is an example to make a dictionary with each item being a pair of a number and its square.

```
squares = {x: x*x for x in range(6)}

# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
print(squares)
```

A dictionary comprehension can optionally contain more for or if statements. An optional if statement can filter out items to form the new dictionary. Here are some examples to make dictionary with only odd items.

Other Dictionary Operations

Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword `in`. Notice that membership test is for keys only, not for values.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True
print(1 in squares)

# Output: True
print(2 not in squares)

# membership tests for key only not value
# Output: False
print(49 in squares)
```

Iterating Through a Dictionary

Using a for loop we can iterate through each key in a dictionary.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
    print(squares[i])
```

Built-in Functions with Dictionary

Built-in functions like `all()`, `any()`, `len()`, `cmp()`, `sorted()` etc. are commonly used with dictionary to perform different tasks. Built-in Functions with Dictionary

Function	Description
<code>all()</code>	Return True if all keys of the dictionary are true (or if the dictionary is empty).
<code>any()</code>	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
<code>len()</code>	Return the length (the number of items) in the dictionary.
<code>cmp()</code>	Compares items of two dictionaries.
<code>sorted()</code>	Return a new sorted list of keys in the dictionary.

Here are some examples that uses built-in functions to work with dictionary.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: 5
print(len(squares))

# Output: [1, 3, 5, 7, 9]
print(sorted(squares))
```

Python Operators

What are operators in python?

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

For example:

```
>>> 2+3
5
```

Here, + is the operator that performs addition. 2 and 3 are the operands and 5 is the output of the operation.

Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Operator	Meaning	Example
+	Add two operands or unary plus	x + y +2
-	Subtract right operand from the left or unary minus	x - y -2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Example : Arithmetic operators in Python

```

x = 15
y = 4
# Output: x + y = 19
print('x + y =',x+y)
# Output: x - y = 11
print('x - y =',x-y)
# Output: x * y = 60
print('x * y =',x*y)
# Output: x / y = 3.75
print('x / y =',x/y)
# Output: x // y = 3
print('x // y =',x//y)
# Output: x ** y = 50625
print('x ** y =',x**y)

```

When you run the program, the output will be:

```

x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625

```

Comparison operators

Comparison operators are used to compare values. It either returns True or False according to the condition.

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	<code>x > y</code>
<	Less than - True if left operand is less than the right	<code>x < y</code>
==	Equal to - True if both operands are equal	<code>x == y</code>
!=	Not equal to - True if operands are not equal	<code>x != y</code>
>=	Greater than or equal to - True if left operand is greater than or equal to the right	<code>x >= y</code>
<=	Less than or equal to - True if left operand is less than or equal to the right	<code>x <= y</code>

Example : Comparison operators in Python

```

x = 10
y = 12
# Output: x > y is False
print('x > y is',x>y)
# Output: x < y is True
print('x < y is',x<y)

```

```
# Output: x == y is False
print('x == y is',x==y)
# Output: x != y is True
print('x != y is',x!=y)
# Output: x >= y is False
print('x >= y is',x>=y)
# Output: x <= y is True
print('x <= y is',x<=y)
```

Logical operators

Logical operators are the and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Example : Logical Operators in Python

```
x = True
y = False
# Output: x and y is False
print('x and y is',x and y)
# Output: x or y is True
print('x or y is',x or y)
# Output: not x is False
print('not x is',not x)
```

The truth table for and is given below:

Truth table for *and*

A | B | A and B -- | -- | --- True | True | True True | False | False False | True | False False | False | False

The truth table for or is given below:

Truth table for or

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

The truth table for *not* is given below:

Truth tabel for not

A	not A
True	False
False	True

some example of their usage are given below

```
>>> True and False
False
>>> True or False
True
>>> not False
True
```

Bitwise operators

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Bitwise operators in Python

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
	Bitwise OR	x y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

Assignment operators

Assignment operators are used in Python to assign values to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

Assignment operators in Python

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
=	x	= 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Special operators

Python language offers some special type of operators like the identity operator or the membership operator. They are described below with examples. Identity operators

is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Identity operators in Python

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Example : Identity operators in Python

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]
# Output: False
print(x1 is not y1)
```

```
# Output: True
print(x2 is y2)
# Output: False
print(x3 is y3)
```

Here, we see that x1 and y1 are integers of same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).

But x3 and y3 are list. They are equal but not identical. It is because interpreter locates them separately in memory although they are equal. Membership operators

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Example : Membership operators in Python

```
x = 'Hello world'
y = {1:'a', 2:'b'}
# Output: True
print('H' in x)
# Output: True
print('hello' not in x)
# Output: True
print(1 in y)
# Output: False
print('a' in y)
```

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

Python Printout

The printout in python can be done with a print() function, with keyword arguments to replace most of the special syntax of the old print statement.

The print statement can be used in the following ways :

```
print("Good Morning")
print("Good", <Variable Containing the String>)
```

```
print("Good" + <Variable Containing the String>)  
print("Good %s" % <variable containing the string>)
```

In Python, **single, double and triple** quotes are used to denote a string. Most use single quotes when declaring a single character. Double quotes when declaring a line and triple quotes when declaring a paragraph/multiple lines.

Double Quotes Use

```
print("Python is very simple language")
```

Output: Python is very simple language

Single Quotes Use

```
print('Hello')
```

Output: Hello

Triple Quotes Use

```
print("""Python is very popular language.  
It is also friendly language.""")
```

Output: Python is very Popular Language. It is also friendly language.

Variable Use

Strings can be assigned to variable say string1 and string2 which can called when using the print statement.

```
str1 = 'Wel'  
print(str1, 'come')
```

Output: Wel come

```
str1 = 'Welcome'  
str2 = 'Python'  
print(str1, str2)
```

Output: Welcome Python

String Concatenation

String concatenation is the "addition" of two strings. Observe that while concatenating there will be no space between the strings.

```
str1 = 'Python'  
str2 = ':'  
print('Welcome' + str1 + str2)
```

Output: WelcomePython:

Using as String

%s is used to refer to a variable which contains a string.

```
str1 = 'Python'  
print("Welcome %s" % str1)
```

Output: Welcome Python

Using other data types

Similarly, when using other data types

%d | Integer %e | exponential %f | Float %o | Octal %x | Hexadecimal

This can be used for conversions inside the print statement itself.

Using as Integer

```
print("Actual Number = %d" %15)
```

Output: Actual Number = 15

Using as Exponential

```
print("Exponential equivalent of the number = %e" %15)
```

Output: Exponential equivalent of the number = 1.500000e+01

Using as Float

```
print("Float of the number = %f" %15)
```

```
Output: Float of the number = 15.000000
```

Using as Octal

```
print("Octal equivalent of the number = %o" %15)
```

```
Output: Octal equivalent of the number = 17
```

Using as Hexadecimal

```
print("Hexadecimal equivalent of the number = %x" %15)
```

```
Output: Hexadecimal equivalent of the number = f
```

Using multiple variables

When referring to multiple variables parenthesis is used.

```
str1 = 'World'  
str2 = ':'  
print("Python %s %s" %(str1, str2))
```

```
Output: Python World :
```

Other Examples of Print Statement:

The following are other different ways the print statement can be put to use.

Example-1:

```
% is used for %d type word  
print("Welcome to %%Python %s" %'language')
```

```
Output: Welcome to %Python language
```

Example-2:

```
\n is used for Line Break.
```

```
print("Sunday\nMonday\nTuesday\nWednesday\nThursday\nFriday\nSaturday")
```

Output:

```
Sunday Monday Tuesday Wednesday Thursday Friday Saturday
```

Example-3:

Any word print multiple times.

```
print('-w3r'*5)
```

```
Output: -w3r-w3r-w3r-w3r-w3r
```

Example-4:

```
\t is used for tab.  
  
print("""  
Language:  
\t1 Python  
\t2 Java\n\t3 JavaScript  
""")
```

```
Output: Language: 1 Python 2 Java 3 JavaScript
```

Precision Width and Field Width

Field width is the width of the entire number and precision is the width towards the right. One can alter these widths based on the requirements.

The default Precision Width is set to 6.

Example-1

Notice upto 6 decimal points are returned. To specify the number of decimal points, '%(fieldwidth).(precisionwidth)f' is used.

```
print("%f" % 5.1234567890)
```

```
Output: 5.123457
```

Example-2

Notice upto 5 decimal points are returned

```
print("%.5f" % 5.1234567890)
```

Output: 5.12346

Example-3

If the field width is set more than the necessary than the data right aligns itself to adjust to the specified values.

```
print("%9.5f" % 5.1234567890)
```

Output: 5.12346

Example-4

Zero padding is done by adding a 0 at the start of fieldwidth.

```
print("%015.5f" % 5.1234567890)
```

Output: 000000005.12346

Example-5

For proper alignment, a space can be left blank in the field width so that when a negative number is used, proper alignment is maintained.

```
print("% 9f" % 5.1234567890)  
print("% 9f" % -5.1234567890)
```

Output: 5.123457 -5.123457

Example-6

'+' sign can be returned at the beginning of a positive number by adding a + sign at the beginning of the field width.

```
print("%+9f" % 5.1234567890)  
print("% 9f" % -5.1234567890)
```

Output: +5.123457 -5.123457

Example-7

As mentioned above, the data right aligns itself when the field width mentioned is larger than the actual field width. But left alignment can be done by specifying a negative symbol in the field width.

```
print("%-9.4f" % 5.1234567890)
```

Output: 5.1235

Python Command Line Input

Python allows for command line input. That means we are able to ask the user for input. The method is a bit different in Python 3.6 than Python 2.7.

- Python 3.6 uses the `input()` method.
- Python 2.7 uses the `raw_input()` method.

The following example asks for the user's name, and when you entered the name, the name gets printed to the screen:

Python 3.6

```
print("Enter your name:")  
x = input()  
print("Hello ", x)
```

```
Python 2.7  
print("Enter your name:")  
x = raw_input()  
print("Hello ", x)
```

Save this file as `demo_input.py`, and load it through the command line/powershell: `C:\Users\Your Name>python demo_input.py`

Our program will prompt the user for a string: Enter your name:

The user now enters a name: Alex

Then, the program prints it to screen with a little message:

Hello Alex

Python Operators, Conditions and If statements

Python Arithmetic, Comparison and Logical Operators

Arithmetic Operators

Operator	Name	Example	Result
+	Addition	x+y	Sum of x and y.
-	Subtraction	x-y	Difference of x and y.
*	Multiplication	x*y	Product of x and y.
/	Division	x/y	Quotient of x and y.
%	Modulus	x%y	Remainder of x divided by y.
**	Exponent	x**y	x**y will give x to the power y
//	Floor Division	x/ y	The division of operands where the result is the quotient in which the digits after the decimal point are removed.

These conditions can be used in several ways, most commonly in "if statements" and loops.

Comparison Operators

Operator	Name	Example	Result
==	Equal	x==y	True if x is exactly equal to y.
!=	Not equal	x!=y	True if x is exactly not equal to y.
>	Greater than	x>y	True if x (left-hand argument) is greater than y (right-hand argument).
<	Less than	x<y	True if x (left-hand argument) is less than y (right-hand argument).
>=	Greater than or equal to	x>=y	True if x (left-hand argument) is greater than or equal to y (left-hand argument).
<=	Less than or equal to	x<=y	True if x (left-hand argument) is less than or equal to y (right-hand argument).

Logical Operators

Operator	Example	Result
and	(x and y)	is True if both x and y are true.
or	(x or y)	is True if either x or y is true.
not	(x not y)	If a condition is true then Logical not operator will make false.

Assignment Operators

Operator	Shorthand	Expression	Description
+=	x+=y	x = x + y	Adds 2 numbers and assigns the result to left operand.
-=	x-= y	x = x -y	Subtracts 2 numbers and assigns the result to left operand.
=	x= y	x = x*y	Multiplies 2 numbers and assigns the result to left operand.
/=	x/= y	x = x/y	Divides 2 numbers and assigns the result to left operand.
%=	x%= y	x = x%y	Computes the modulus of 2 numbers and assigns the result to left operand.
=	x=y	x = x**y	Performs exponential (power) calculation on operators and assign value to the equivalent to left operand.
//=	x//=y	x = x//y	Performs floor division on operators and assign value to the left operand.

Conditional Operators

Conditional expressions or ternary operator have the lowest priority of all Python operations. The expression `x if C else y` first evaluates the condition, `C` (not `x`); if `C` is true, `x` is evaluated and its value is returned; otherwise, `y` is evaluated and its value is returned.

if..elif..else

The if-elif-else statement is used to conditionally execute a statement or a block of statements. Conditions can be true or false, execute one thing when the condition is true, something else when the condition is false

if statement

The Python if statement is same as it is with other programming languages. It executes a set of statements conditionally, based on the value of a logical expression.

Syntax:

```
if expression :
    statement_1
    statement_2
    ....
```

In the above case, 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

if .. else statement

In Python if .. else statement, if has two blocks, one following the expression and other following the else clause. Here is the syntax.

Syntax:

```
if expression :  
    statement_1  
    statement_2  
    ....  
else :  
    statement_3  
    statement_4  
    ....
```

In the above case 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. See the following example. The program will print the second print statement as the value of a is 10.

```
a=10  
if(a>10):  
    print("Value of a is greater than 10")  
else :  
    print("Value of a is 10")
```

if .. elif .. else statement

Sometimes a situation arises when there are several conditions. To handle the situation Python allows adding any number of elif clause after an if and before an else clause. Here is the syntax.

Syntax:

```
if expression1 :  
    statement_1  
    statement_2  
    ....  
elif expression2 :  
    statement_3  
    statement_4  
    ....  
elif expression3 :  
    statement_5  
    statement_6  
    .....  
else :  
    statement_7  
    statement_8
```

In the above case 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. In the following example, we have applied if, series of elif and else to get the type of a variable.

```
var1 = 1+2j
if (type(var1) == int):
    print("Type of the variable is Integer")
elif (type(var1) == float):
    print("Type of the variable is Float")
elif (type(var1) == complex):
    print("Type of the variable is Complex")
elif (type(var1) == bool):
    print("Type of the variable is Bool")
elif (type(var1) == str):
    print("Type of the variable is String")
elif (type(var1) == tuple):
    print("Type of the variable is Tuple")
elif (type(var1) == dict):
    print("Type of the variable is Dictionaries")
elif (type(var1) == list):
    print("Type of the variable is List")
else:
    print("Type of the variable is Unknown")
```

Output:

```
Type of the variable is Complex
```

Nested if .. else statement

In general nested if-else statement is used when we want to check more than one conditions. 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. Here is the syntax :

Syntax:

```
if expression1 :
    if expression2 :
        statement_3
        statement_4
    ....
else :
    statement_5
    statement_6
    ....
else :
```

```
statement_7
statement_8
```

In the above syntax expression1 is checked first, if it evaluates to true then the program control goes to next if - else part otherwise it goes to the last else statement and executes statement_7, statement_8 etc.. Within the if - else if expression2 evaluates true then statement_3, statement_4 will execute otherwise statement_5, statement_6 will execute. See the following example.

```
age = 38
if (age >= 11): print("You are eligible to see the Football match.")
if (age <= 20 or age >= 60):
    print("Ticket price is $12")
else: print("Tic kit price is $20")
else: print("You're not eligible to buy a ticket.")
```

Output :

```
You are eligible to see the Football match. Tic kit price is $20
```

In the above example age is set to 38, therefore the first expression (age >= 11) evaluates to True and the associated print statement prints the string "You are eligible to see the Football match". There after program control goes to next if statement and the condition (38 is outside <=20 or >=60) is matched and prints "Tic ket price is \$12".

Python Loops

Loops are used to repeatedly execute a block of program statements. In Python, and in most of programming languages, there are two main types of loops: for loop and while loops.

For loop

In Python for loop is used to iterate over the items of any sequence including the Python list, string, tuple etc. The for loop is also used to access elements from a container (for example list, string, tuple) using built-in function range().

The for loop is also used to access elements from a container (for example list, string, tuple) using built-in function range().

Syntax:

```
for variable_name in sequence :
    statement_1
    statement_2
    ....
```

Parameter:

Name	Description
variable_name	It indicates target variable which will set a new value for each iteration of the loop.

Name	Description
sequence	A sequence of values that will be assigned to the target variable <code>variable_name</code> . Values are provided using a list or a string or from the built-in function <code>range()</code> .

`statement_1`,
`statement_2 ...` | Block of program statements.

Example: Python for loop

```
#The list has four elements, indices start at 0 and end at 3
color_list = ["Red", "Blue", "Green", "Black"]
for c in color_list:
    print(c)
```

```
Red
Blue
Green
Black
```

In the above example `color_list` is a sequence contains a list of various color names. When the for loop executed the first item (i.e. Red) is assigned to the variable `c`. After this, the print statement will execute and the process will continue until we reach the end of the list.

Python for loop and `range()` function

The `range()` function returns a list of consecutive integers. The function has one, two or three parameters where last two parameters are optional. It is widely used in for loops. Here is the syntax.

`range(a)` : Generates a sequence of numbers from 0 to `a`, excluding `a`, incrementing by 1.

```
range(a)
range(a, b)
range(a, b, c)
```

Syntax:

for in `range()`:

Example:

```
for a in range(4):
    print(a)
```

Output: 0 1 2 3

range(a,b): Generates a sequence of numbers from a to b excluding b, incrementing by 1.

Syntax:

for "variable" in range("start_number", "end_number"):

Example:

```
for a in range(2,7):  
    print(a)
```

Output: 2 3 4 5 6

range(a,b,c): Generates a sequence of numbers from a to b excluding b, incrementing by c.

Example:

```
for a in range(2,19,5):  
    print(a)
```

Output: 2 7 12 17

Python for loop: Iterating over tuple, list, dictionary

Example: Iterating over tuple

The following example counts the number of even and odd numbers from a series of numbers.

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9) # Declaring the tuple  
count_odd = 0  
count_even = 0  
for x in numbers:  
    if x % 2:  
        count_odd+=1  
    else:  
        count_even+=1  
print("Number of even numbers :",count_even)  
print("Number of odd numbers :",count_odd)
```

Output: Number of even numbers:4 Number of odd numbers: 5

In the above example a tuple named numbers is declared which holds the integers 1 to 9.

The best way to check if a given number is even or odd is to use the modulus operator (%). The operator returns the remainder when dividing two numbers. Modulus of 8 % 2 returns 0 as 8 is divided by 2, therefore 8 is even and modulus of 5 % 2 returns 1 therefore 5 is odd.

The for loop iterates through the tuple and we test modulus of $x \% 2$ is true or not, for every item in the tuple and the process will continue until we reach the end of the tuple. When it is true count_even increase by one otherwise count_odd is increased by one. Finally, we print the number of even and odd numbers through print statements.

Example: Iterating over list

In the following example for loop iterates through the list "datalist" and prints each item and its corresponding Python type.

```
datalist = [1452, 11.23, 1+2j, True, 'w3resource', (0, -1), [5, 12],  
{ "class": 'V', "section": 'A' }]  
for item in datalist:  
    print ("Type of ",item, " is ", type(item))
```

Output: Type of 1452 is <class 'int'> Type of 11.23 is <class 'float'> Type of (1+2j) is <class 'complex'>
Type of True is <class 'bool'> Type of w3resource is <class 'str'> Type of (0, -1) is <class 'tuple'> Type
of [5, 12] is <class 'list'> Type of {'section': 'A', 'class': 'V'} is <class 'dict'>

Example: Iterating over dictionary

In the following example for loop iterates through the dictionary "color" through its keys and prints each key.

```
color = {"c1": "Red", "c2": "Green", "c3": "Orange"}  
for key in color:  
    print(key)
```

c2 c1 c3

Following for loop iterates through its values :

```
color = {"c1": "Red", "c2": "Green", "c3": "Orange"}  
for value in color.values():  
    print(value)
```

Green Red Orange

You can attach an optional else clause with for statement, in this case, syntax will be -

```
for variable_name in sequence :  
    statement_1  
    statement_2  
    ....  
else :  
    statement_3
```

```
statement_4
....
```

The else clause is only executed after completing the for loop. If a break statement executes in first program block and terminates the loop then the else clause does not execute.

While loop

While loops are used to repeatedly execute a block of program statements. Here is the syntax.

Syntax:

```
while (expression) :
    statement_1
    statement_2
    ....
```

The while loop runs as long as the expression (condition) evaluates to True and execute the program block. The condition is checked every time at the beginning of the loop and the first time when the expression evaluates to False, the loop stops without executing any remaining statement(s). The following example prints the digits 0 to 4 as we set the condition $x < 5$.

```
x = 0;
while (x < 5):
    print(x)
    x += 1
```

Output: 0 1 2 3 4

One thing we should remember that a while loop tests its condition before the body of the loop (block of program statements) is executed. If the initial test returns false, the body is not executed at all. For example the following code never prints out anything since before executing the condition evaluates to false.

```
x = 10;
while (x < 5):
    print(x)
    x += 1
```

Python while loop

The following while loop is an infinite loop, using True as the check condition:

```
x = 10; while (True): print(x) x += 1
```

Python while loop is infinite unless the *!!check condition!!* is changed to boolean False

Python: while and else statement

There is a structural similarity between while and else statement. Both have a block of statement(s) which is only executed when the condition is true. The difference is that block belongs to if statement executes once whereas block belongs to while statement executes repeatedly.

You can attach an optional else clause with while statement, in this case, syntax will be -

```
while (expression) :  
    statement_1  
    statement_2  
    .....  
else :  
    statement_3  
    statement_4  
    .....
```

The while loop repeatedly tests the expression (condition) and, if it is true, executes the first block of program statements. The else clause is only executed when the condition is false it may be the first time it is tested and will not execute if the loop breaks, or if an exception is raised. If a break statement executes in first program block and terminates the loop then the else clause does not execute. In the following example, while loop calculates the sum of the integers from 0 to 9 and after completing the loop, else statement executes.

```
x = 0;  
s = 0  
while (x < 10):  
    s = s + x  
    x = x + 1  
else :  
    print('The sum of first 9 integers : ',s)
```

Output: The sum of first 9 integers: 45

Python while else loop

Example: while loop with if-else and break statement

```
x = 1;  
s = 0  
while (x < 10):  
    s = s + x  
    x = x + 1  
    if (x == 5):  
        break  
else :  
    print('The sum of first 9 integers : ',s)  
print('The sum of ',x,' numbers is :',s)
```

```
Output: The sum of 5 numbers is : 10
```

In the above example the loop is terminated when x becomes 5. Here we use break statement to terminate the while loop without completing it, therefore program control goes to outside the while - else structure and execute the next print statement.

break statement

The break statement is used to exit a for or a while loop. The purpose of this statement is to end the execution of the loop (for or while) immediately and the program control goes to the statement after the last statement of the loop. If there is an optional else statement in while or for loop it skips the optional clause also. Here is the syntax.

Syntax:

```
while (expression1) :  
    statement_1  
    statement_2  
    .....  
    if expression2 :  
        break  
  
for variable_name in sequence :  
    statement_1  
    statement_2  
    if expression3 :  
        break
```

Example: break in for loop

In the following example for loop breaks when the count value is 5. The print statement after the for loop displays the sum of first 5 elements of the tuple numbers.

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9) # Declaring the tuple  
num_sum = 0  
count = 0  
for x in numbers:  
    num_sum = num_sum + x  
    count = count + 1  
    if count == 5:  
        break  
print("Sum of first ",count,"integers is: ", num_sum)
```

```
Output: Sum of first 5 integers is: 15
```

Example: break in while loop

In the following example while loop breaks when the count value is 5. The print statement after the while loop displays the value of num_sum (i.e. 0+1+2+3+4).

```
num_sum = 0
count = 0
while(count<10):
    num_sum = num_sum + count
    count = count + 1
    if count== 5:
        break
print("Sum of first ",count,"integers is: ", num_sum)
```

Output: Sum of first 5 integers is : 10

continue statement

The `continue` statement is used in a `while` or `for` loop to take the control to the top of the loop without executing the rest statements inside the loop. Here is a simple example.

```
for x in range(7):
    if (x == 3 or x==6):
        continue
    print(x)
```

Output: 0 1 2 4 5

In the above example, the for loop prints all the numbers from 0 to 6 except 3 and 6 as the continue statement returns the control of the loop to the top

Collections

Note* : Please go over the data types before reading this material

Python Loops

Python for Loop

What is for loop in Python?

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal. Syntax of for Loop

```
for val in sequence:
    Body of for
```

Here, `val` is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Example: Python for Loop

```
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

# Output: The sum is 48
print("The sum is", sum)
```

The range() function

We can generate a sequence of numbers using `range()` function. `range(10)` will generate numbers from 0 to 9 (10 numbers). We can also define the start, stop and step size as `range(start,stop,step size)`. step size defaults to 1 if not provided. This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go. To force this function to output all the items, we can use the function `list()`. The following example will clarify this.

```
# Output: range(0, 10)
print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(range(10)))

# Output: [2, 3, 4, 5, 6, 7]
print(list(range(2, 8)))

# Output: [2, 5, 8, 11, 14, 17]
print(list(range(2, 20, 3)))
```

We can use the `range()` function in for loops to iterate through a sequence of numbers. It can be combined with the `len()` function to iterate through a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index
for i in range(len(genre)):
    print("I like", genre[i])
```

When you run the program, the output will be:

```
I like pop
I like rock
I like jazz
```

for loop with else

A for loop can have an optional **else** block as well. The else part is executed if the items in the sequence used in for loop exhausts. **break** statement can be used to stop a for loop. In such case, the else part is ignored. Hence, a for loop's else part runs if no break occurs. Here is an example to illustrate this.

```
digits = [0, 1, 5]

for i in digits:
    print(i)
else:
    print("No items left.")
```

When you run the program, the output will be:

```
0
1
5
No items left.
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints

```
No items left.
```

Python while Loop

What is while loop in Python?

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true. We generally use this loop when we don't know beforehand, the number of times to iterate.

Syntax of while Loop in Python:

```
while test_expression:
    Body of while
```

In while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

In Python, the body of the while loop is determined through indentation. Body starts with indentation and the first unindented line marks the end. Python interprets any non-zero value as True. None and 0 are interpreted as False

```
# Program to add natural
# numbers upto
# sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

When you run the program, the output will be:

```
Enter n: 10
The sum is 55
```

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n.

We need to increase the value of counter variable in the body of the loop. This is very important. Failing to do so will result in an infinite loop (loop that goes on forever).

Finally the result is displayed

while loop with else

Same as that of for loop, we can have an optional else block with while loop as well. The **else** part is executed if the condition in the while loop evaluates to False. The while loop can be terminated with a break statement. In such case, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false. Here is an example to illustrate this.

```
# Example to illustrate
# the use of else statement
# with the while loop

counter = 0

while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Output

```
Inside loop
Inside loop
Inside loop
Inside else
```

Here, we use a counter variable to print the string Inside loop three times. On the forth iteration, the condition in while becomes False. Hence, the else part is executed

Python break and continue

What is the use of break and continue in Python?

In Python, break and continue statements can alter the flow of a normal loop. Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The **break** and **continue** statements are used in these cases.

Python break statement

The **break** statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If break statement is inside a nested loop (loop inside another loop),

break will terminate the innermost loop. Syntax of break

```
break
```

Example: Python break

```
for val in "string":  
    if val == "i":  
        break  
    print(val)  
  
print("The end")
```

Output

```
s  
t  
r  
The end
```

In this program, we iterate through the "string" sequence. We check if the letter is "i", upon which we break from the loop. Hence, we see in our output that all the letters up till "i" gets printed. After that, the loop terminates.

Python continue statement

The **continue** statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration. Syntax of Continue

```
continue
```

Example: Python continue

```
for val in "string":  
    if val == "i":  
        continue  
    print(val)  
  
print("The end")
```

Output

```
s
t
r
n
g
The end
```

This program is same as the above example except the break statement has been replaced with continue. We continue with the loop, if the string is "i", not executing the rest of the block. Hence, we see in our output that all the letters except "i" gets printed.

Python pass statement

What is pass statement in Python?

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.

However, nothing happens when pass is executed. It results into no operation (NOP). Syntax of pass

```
pass
```

We generally use it as a placeholder.

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the pass statement to construct a body that does nothing.

```
# pass is just a placeholder for
# functionality to be added later.
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

Python Arrays

In programming, an array is a collection of elements of the same type. Arrays are popular in most programming languages like Java, C/C++, JavaScript and so on. However, in Python, they are not that common. When people talk about Python arrays, more often than not, they are talking about Python lists. If you don't know what lists are, you should definitely check Python list article.

That being said, array of numeric values are supported in Python by the array module.

Python Lists Vs array Module as Arrays

We can treat lists as arrays. However, we cannot constrain the type of elements stored in a list. For example:

```
a = [1, 3.5, "Hello"]
```

If you create arrays using the array module, all elements of the array must be of the same numeric type.

```
import array as arr
a = arr.array('d', [1, 3.5, "Hello"]) // Error
```

How to create arrays?

As you might have guessed from the above example, we need to import array module to create arrays. For example:

```
import array as arr
a = arr.array('d', [1.1, 3.5, 4.5])
print(a)
```

Here, we created an array of float type. The letter 'd' is a type code. This determines the type of the array during creation. Commonly used type codes:

Code	C Type	Python Type	Min bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'f'	float	float	4
'd'	double	float	8

We will not discuss about different C types in this article. We will use two type codes in this entire article: 'i' for integers and 'd' for floats.

Note: The 'u' type code for Unicode characters is deprecated since version 3.3. Avoid using it when possible.

How to access array elements?

We use indices to access elements of an array:

```
import array as arr
a = arr.array('i', [2, 4, 6, 8])
print("First element:", a[0])
print("Second element:", a[1])
print("Last element:", a[-1])
```

Remember, the index starts from 0 (not 1) similar to lists.

How to slice arrays?

We can access a range of items in an array by using the slicing operator .:

```
import array as arr
numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
numbers_array = arr.array('i', numbers_list)
print(numbers_array[2:5]) # 3rd to 5th
print(numbers_array[:5]) # beginning to 4th
print(numbers_array[5:]) # 6th to end
print(numbers_array[:]) # beginning to end
```

When you run the program, the output will be:

```
array('i', [62, 5, 42])
array('i', [2, 5, 62])
array('i', [52, 48, 5])
array('i', [2, 5, 62, 5, 42, 52, 48, 5])
```

How to change or add elements?

Arrays are mutable; their elements can be changed in a similar way like lists.

```
import array as arr
numbers = arr.array('i', [1, 2, 3, 5, 7, 10])
# changing first element
numbers[0] = 0
print(numbers)      # Output: array('i', [0, 2, 3, 5, 7, 10])
# changing 3rd to 5th element
numbers[2:5] = arr.array('i', [4, 6, 8])
print(numbers)      # Output: array('i', [0, 2, 4, 6, 8, 10])
```

We can add one item to a list using the `append()` method, or add several items using `extend()` method.

```
import array as arr
numbers = arr.array('i', [1, 2, 3])
numbers.append(4)
print(numbers)      # Output: array('i', [1, 2, 3, 4])
# extend() appends iterable to the end of the array
numbers.extend([5, 6, 7])
print(numbers)      # Output: array('i', [1, 2, 3, 4, 5, 6, 7])
```

We can concatenate two arrays using `+` operator.

```
import array as arr
odd = arr.array('i', [1, 3, 5])
even = arr.array('i', [2, 4, 6])
numbers = arr.array('i') # creating empty array of integer
numbers = odd + even
print(numbers)
```

How to remove/delete elements?

We can delete one or more items from an array using Python's `del` statement.

```
import array as arr
number = arr.array('i', [1, 2, 3, 3, 4])
del number[2] # removing third element
print(number) # Output: array('i', [1, 2, 3, 4])
del number # deleting entire array
print(number) # Error: array is not defined
```

We can use the `remove()` method to remove the given item, and `pop()` method to remove an item at the given index.

```
import array as arr
numbers = arr.array('i', [10, 11, 12, 12, 13])
numbers.remove(12)
print(numbers) # Output: array('i', [10, 11, 12, 13])
print(numbers.pop(2)) # Output: 12
print(numbers) # Output: array('i', [10, 11, 13])
```

Check this page to learn more about Python array and array methods:

When to use arrays?

Lists are much more flexible than arrays. They can store elements of different data types including string. Also, lists are faster than arrays. And, if you need to do mathematical computation on arrays and matrices, you are much better off using something like NumPy library.

Unless you don't really need arrays (array module may be needed to interface with C code), don't use them.

Python Functions

In all programming and scripting language, a function is a block of program statements which can be used repetitively in a program. It saves the time of a developer. In Python concept of function is same as in other languages. There are some built-in functions which are part of Python. Besides that, we can define functions according to our need. User defined function

In Python, a user-defined function's declaration begins with the keyword `def` and followed by the function name. The function may take argument(s) as input within the opening and closing parentheses, just after the function name followed by a colon. After defining the function name and argument(s) a block of program statement(s) start at the next line and these statement(s) must be indented.

Here is the syntax of a user defined function.

Syntax:

```
def function_name(argument1, argument2, ...) :  
    statement_1  
    statement_2  
    ....
```

Call a function

Calling a function in Python is similar to other programming languages, using the function name, parenthesis (opening and closing) and parameter(s). See the syntax, followed by an example. Syntax:

```
function_name(arg1, arg2)
```

Example:

```
def avg_number(x, y):  
    print("Average of ", x, " and ", y, " is ", (x+y)/2)  
avg_number(3, 4)
```

Output:

```
Average of 3 and 4 is 3.5
```

Explanation:

1. Lines 1-2 : Details (definition) of the function.
2. Line 3 : Call the function.
3. Line 1 : Pass parameters : $x = 3$, $y = 4$
4. Line 2 : Print the value of two parameters as well as their average value.

Function without arguments:

The following function has no arguments.

```
def function_name() :  
    statement_1  
    statement_2  
    ....
```

Example:

```
def printt():  
    print("This is Python 3.2 Tutorial")  
    print("This is Python 3.2 Tutorial")  
    print("This is Python 3.2 Tutorial")  
printt()
```

Output:

This is Python 3.2 Tutorial This is Python 3.2 Tutorial This is Python 3.2 Tutorial

Explanation:

1. Lines 1-4 : Details (definition) of the function.
2. Line 5 : Call the function.
3. Line 1 : No parameter passes.
4. Line 2-4 : Execute three print statements.

The Return statement in function

In Python the return statement (the word return followed by an expression.) is used to return a value from a function, return statement without an expression argument returns none. See the syntax.

```
def function_name(argument1, argument2, ...) :  
    statement_1  
    statement_2  
    ....  
    return expression  
  
function_name(arg1, arg2)
```


Example:

The following function returns the square of the sum of two numbers.

```
def nsquare(x, y):  
    return (x*x + 2*x*y + y*y)  
print("The square of the sum of 2 and 3 is : ", nsquare(2, 3))
```

Output:

The square of the sum of 2 and 3 is : 25

Explanation:

1. Lines 1-2 : Details (definition) of the function.
2. Line 3 : Call the function within a print statement.
3. Line 1 : Pass the parameters x = 2, y = 3
4. Line 2 : Calculate and return the value of (x + y)²

Default Argument Values

In function's parameters list we can specify a default value(s) for one or more arguments. A default value can be written in the format "argument1 = value", therefore we will have the option to declare or not declare a value for those arguments. See the following example.

Example:

The following function returns the square of the sum of two numbers, where default value of the second argument is 2.

```
def nsquare(x, y = 2):  
    return (x*x + 2*x*y + y*y)  
print("The square of the sum of 2 and 2 is : ", nsquare(2))  
print("The square of the sum of 2 and 3 is : ", nsquare(2, 4))
```

Output:

The square of the sum of 2 and 2 is: 16 The square of the sum of 2 and 4 is : 36

Explanation:

1. Lines 1-2 : Details (definition) of the function. For first print statement [Line no 3]
2. Line 3 : Call the function without a second argument, within a print statement.
3. Line 1 : Pass the parameters x = 2, default value.
4. Line 2 : Calculate and return the value of (x + y)² For second print statement [Line no 4]
5. Line 3 : Call the function with all arguments, within a print statement.
6. Line 1 : Pass the parameters x = 2, y = 4.

7. Line 2 : Calculate and return the value of $(x + y)^2$

Keyword Arguments:

We have already learned how to use default arguments values, functions can also be called using keyword arguments. Arguments which are preceded with a variable name followed by a '=' sign (e.g. `var_name=`) are called keyword arguments.

All the keyword arguments passed must match one of the arguments accepted by the function. You may change the order of appearance of the keyword. See the following example.

Example:

```
def marks(english, math = 85, science = 80):
    print('Marks in : English is - ', english, ', Math - ', math, ', Science
    - ', science)

marks(71, 77)
marks(65, science = 74)
marks(science = 70, math = 90, english = 75)
```

Output:

Marks in : English is - 71 , Math - 77 , Science - 80 Marks in : English is - 65 , Math - 85 , Science - 74 Marks in : English is - 75 , Math - 90 , Science - 70

Explanation:

Line 1: The function named marks has three parameters, there is no default value in the first parameter (english) but remaining parameters have default values (math = 85, science = 80). Line 3: The parameter english gets the value of 71, math gets the value 77 and science gets the default value of 80. Line 4: The parameter english gets the value of 65, math gets the default value of 85 and science gets the value of 74 due to keyword arguments. Line 5: Here we use three keyword arguments and the parameter english gets the value 75, math gets the value 90 and science gets the value 70.

Arbitrary Argument Lists:

The **arbitrary argument list** is another way to pass arguments to a function. In the function body, these arguments will be wrapped in a tuple and it can be defined with `*args` construct. Before this variable, you can define a number of arguments or no argument.

Example:

```
def sum(*numbers):
    s = 0
    for n in numbers:
        s += n
    return s
```

```
print(sum(1,2,3,4))
```

Output:

10

Lambda Forms:

In Python, small anonymous (unnamed) functions can be created with lambda keyword. Lambda forms can be used as an argument to other function where function objects are required but syntactically they are restricted to a single expression. A function like this:

```
def average(x, y):  
    return (x + y)/2  
  
print(average(4, 3))
```

may also be defined using lambda

```
print((lambda x, y: (x + y)/2)(4, 3))
```

Output:

3.5

Python Documentation Strings

In Python, a string literal is used for documenting a module, function, class, or method. You can access string literals by **doc** (notice the double underscores) attribute of the object (e.g. my_function.doc).

Docstring Conventions :

- String literal literals must be enclosed with a triple quote. Docstring should be informative
- The first line may briefly describe the object's purpose. The line should begin with a capital letter and ends with a dot.
- If a documentation string is a multi-line string then the second line should be blank followed by any detailed explanation starting from the third line.

See the following example with multi-line docstring.

```
def avg_number(x, y):  
    """Calculate and Print Average of two Numbers.  
  
    Created on 29/12/2012. python-docstring-example.py
```

```
print("Average of ",x," and ",y, " is ",(x+y)/2)
```

Try and Except

In exception handling in Python, we use the try and except statements to catch and handle exceptions. The code within the try clause is executed statement by statement.

If an exception occurs, the rest of the try block is skipped and the except clause is executed.

```
try:
    'apple' + 6
except Exception:
    print "Cannot concatenate 'str' and 'int' objects"
```

OUTPUT

```
Cannot concatenate 'str' and 'int' objects
```

We avoid the traceback error message elegantly with a simple message like above by using try except statements for exception handling.

In addition to using an except block after the try block, we can also use the finally block. The finally clause is optional. It is intended to define clean-up actions that must be executed under all circumstances

A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.

Actions, like closing a file, GUI or disconnecting from the network, are performed in the finally clause to guarantee execution.

Here is an example of file operations to illustrate finally statement.

```
try:
    f = open("foo.txt",encoding = 'utf-8')
    # perform file operations
finally:
    f.close()
```

This type of statement makes sure that the file is closed whether an exception occurs or not.