Build applications in Python the anti textbook

Suraj

Build applications in Python the anti textbook

Suraj

This book is for sale at http://leanpub.com/antitextbookpy

This version was published on 2019-11-02

Contents

Introduction

Watch on YouTube[1]

Python is an open source[2], cross platform, interpreted language. Cross platform means that a program written on any platform is guaranteed to run across any platform which Python supports. It is a powerful language because there are many third party packages available. The official repository of those packages is http://pypi.python.org. Python comes with a tool called `pip` which allows downloading libraries from the official repository.

Python is widely used across major domains. YouTube, Quora, Hulu, Dropbox are just a few platforms written in Python. We can write command line applications, bootloaders, Robots, Machine Learning algorithms, automate test cases, system administration automation etc using Python.

There are few things which make Python an awesome language:

1. English like syntax: It takes very little time to learn the language and write code. This makes it an excellent prototyping language.
2. Less number of lines: Programs implemented in Python have significantly less number of lines as compared to other languages.
3. High level data structures: Hashmaps, sets, lists make it really easy to write complex programs with comparatively less effort..

Drawbacks:

1. Speed: Since it is an interpreted language, it is slower than most languages.
2. Debugging: Being a dynamically typed language, it is somewhat difficult to debug the programs.

Python 2 vs Python3

Python3 is the successor of Python2. In 2020 Python2 will be history. This tutorial is based on Python3 as it is the present and future of the language. Python3 is a backwards incompatible with Python2, which means, code written for Python2 is not guaranteed to run on Python3. There is a way of writing code which runs both on Python 2 and 3, but it is beyond the scope of this guide.

[1]https://www.youtube.com/watch?v=7wuKDDMb3R4
[2]https://github.com/python/cpython

Installation

1. Windows: Download the latest .exe file of Python3 from https://python.org and click on Next Next.
2. Android: Install Termux (https://termux.com/help.html), and then `apt-get install python3`..
3. Linux: sudo apt-get install python3 / sudo yum install python3 / use other package manager
4. Mac: brew install python3.
5. iOS: python 3 for ios (not free).

Links

|Next[3] | Previous[4] | Index[5] | —- | —- | —- |

[3] 02-more-about-language.md
[4] ../README.md
[5] ../SUMMARY.md

How to run Python code?

Watch on YouTube[6]

There are two modes of running Python code: Interactive and Batch.

Interactive mode

Code is typed inside an interpreter session which gets evaluated immediately. This mode is suitable for small edits.

To start the interpreter, type `python3` or `python` on your terminal or command prompt.

python3 vs python

You have to figure out what to use while invoking the interpreter (python3 or python) based on the following:

Type `python --version` on your command prompt.

```
If it says Python 2, then you'll need to install python3 and execute code using `pyt\
hon3 file.py`.
If it says Python 3, then you'll need to run all the code in this book as `python fi\
le.py`
```

The Interpreter looks like this when it starts:

```
    Python 3.6.0 (default, Jan 13 2017, 22:22:15)
    Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you see `>>>`, this means that the interpreter has started and is waiting for your input. Type `print("Hello Python!")`.

```
>>> print("Hello Python!")
Hello Python!
```

The interpreter evaluates each line which you type. The `print` function is used to print content on the terminal. This is why you saw "Hello Python!" printed immediately on the next line.

Using the Interpreter

Start a new interpreter session and type `1 + 1` and hit enter. You will see the below output.

[6] https://www.youtube.com/watch?v=wSqRUTS7uAg

```
>>> 1 + 1
2
```

Here, `1 + 1` is called an expression and `2` is the output.

`+` will calculate the sum of two numbers (1 and 1) and return the result.

```
>>> 1 - 1
0
```

`-` will calculate the subtraction of `1` and `1` and return the result.

```
>>> 10 * 10
100
```

`*` multiplies 10 with 10 and return the result.

```
>>> 10 ** 2
100
```

`**` is called the power operator, it returns `10` to the power of `2`.

Operator Types

In the above expressions, `+`, `-`, `*` are called operators, they operate on numbers (which are called operands).

In general, there are three types of operators:

1. Unary operators: Require one operand to operate on.
2. Binary operators: Require two operands to operate on. Most of the operators are binary (`-`, `+`, `*`, `**` etc)
3. Ternary operators: Require three operators to operate on.

We won't be looking at unary and ternary operators now, we need to understand the language before jumping to ternary operators.

Batch Mode

The interpreter makes it difficult to write big programs. The best you can do in an interpreter is type lines of code. But, when you want to write programs and build applications, it is better to use the batch mode. In this, you create a text file who's name ends with `.py` like `file.py` and save it to your machine on some folder. We then run the file as `python3 file.py`.

To execute the program, you'd need to use the `cd` program to change folders.

For this book, create a folder. Open the terminal and type this.

Mac/Linux (Applications -> Terminal)

1. `mkdir ~/antitextbookpy` You will save all the code files in this folder. You have to do this only once.
2. `cd ~/antitextbookpy` This will change the current working directory to `~/antitextbookpy`, you have to do this every time you open a new terminal window or tab.

Windows: (open the Command Prompt)

1. `mkdir antitextbookpy`
2. `cd antitextbookpy`

You would need to create a file in this folder, save it by giving it a `.py` extension then run it like `python file.py` from the command line. If it says file not found, then you might have to check if you followed the above steps (of changing directories).

Links

|Next[7] | Previous[8] | Index[9] | ---| ---| ---|

[7]03-01-understanding-variables.md
[8]01-intro-to-python.md
[9]../SUMMARY.md

Variables

Watch on YouTube[10] | Read the docs[11]

Variables are used to store values in memory. When using variables in Python, we do not have to declare the datatype of the variable. Python is a dynamic typing language, which means that when the interpreter will execute the code, it will figure out the data type of the variables on it's own.

Every variable has these things:

1. Name: The variable name has to start with any valid unicode letter except those that have special significance like \ or numbers.
2. Value: The content which is being stored in the variable.
3. Address: The location of the variable in the memory.

There are restrictions about the variable name,

- They can't start with numbers or special characters. We can not name a variable `0b` or `\a` .
- They can not use reserved keywords (try, catch, if, for, while, except, in among others).

If you want to create a variable `i` with the value `1`, type this in the interpreter:

```
>>> i = 1
```

This will create a variable `i` which:

1. Has the value `1`.
2. Is of the data type integer.
3. Is located in some memory location.

Finding address of a variable.

To find out the address of the variable, we use the id()[12] function.

[10] https://www.youtube.com/watch?v=3_-W0S1VdLo
[11] https://docs.python.org/3/reference/expressions.html#atom-identifiers
[12] https://docs.python.org/3/library/functions.html?highlight=id#id

```
>>> print(id(i))
405911019
```

405911019 is the memory location of the variable `i`.

Finding data type of a variable.

`type` is a builtin function which returns the data type of the argument passed to it. Read the docs[13]

```
>>> print(type(1))
<class 'int'>
```

The type() function takes an argument and returns the data type of the argument.

Taking input from the user.

At times, we want to take input from the user, we can use the `input()` function to do so.

```
>>> name = input("enter your name: ")
enter your name: python

>>> print("Your name is ", name)
Your name is  python
```

This will take the name of the user as input and store it in variable `name`. `input` takes a string argument which is the message it should print while taking the input.

By default, `input` returns a string. We have to convert a variable from string to integer by using the `int()` function. Each data type provides a function for data type conversion, for instance, `str()` is used to convert a value from other data type to string.

```
>>> age = input("enter your age: ")
enter your age: 23

>>> age = int(age)

>>> print(type(age))
<class='int'>
```

When we do `age = int(age)`, this is not a syntax error, because the interpreter is going to create a **new** integer variable with the name of age (and delete the old string variable).

[13]https://docs.python.org/3/library/functions.html?highlight=id#type

Exercise

1. Take the user's name, age and height and print it to the terminal.
2. Take a number from the user and print it.

```
>>> i = 'Python'
>>> # Creates a string variable with value Python
>>> print(id(i))
505911019
>>> type(i) # variable i is of type string now.
<class 'string'>
```

Variable types

Numeric

Read the docs[14].

Integer

```
>>> i = 0
>>> i = 10000
>>> i = 123333
>>> print(type(1))
<type 'int'>
```

Float

```
>>> i = 1.1
>>> i = 3.3333344445
>>> print(type(1.0))
<type 'float'>
```

Complex

[14] https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

```
i = complex(1,2) # 1+2j
i = complex(12,23) # 12 + 23j
```

String

Read the docs[15]

```
i = "this string\n\t" # \n and \t are special values if used with "
i = 'this string\n\t' # \n and \t are special values if used with '
i = r'this string\n\t' # \n and \t do not hold special value if prepended with r, r \
stands for raw'
i = ''' multi line string
        which has
        multi lines'''
i = """
        a ridiculously multi line string
        """
```

String validity

The location of the ending quotes doesn't matter, the only thing which matters is that **it should exist**. The value of a string is in between the starting and ending quotes, either single, double or triple.

`'this is a string', "this is a string", "'this is a string", '"this is a string', """ this is 'a string '""", '"""hi'`

All are *valid* strings, they start and end with the same quote. If a string starts and ends with a single quote, then any number of double quotes can be part of the string, the same is true vice versa.

`'"python"` and `"python''` are invalid strings. Both of them do not start and end with the same quote.

Boolean

`True` and `False` are special values in Python3. In previous version of the language, we were allowed to create a variable of name True and False, but now they are reserved.

Read the docs[16]

[15] https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
[16] https://docs.python.org/3/library/stdtypes.html#boolean-values

```
>>> i = True
>>> j = False
>>> print(i)
True
```

Sequence type

Read the docs[17]

List

List stores multiple values of any data type.

```
>>> i = [1,"Linux",3,"bash",[1,2,"sh"]]]
```

Tuples

Read the docs[18]

Tuples are lists from which elements can’t be added/deleted/modified.

```
>>> a = (1,2)
>>> a[0]=2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
a = (1,2,3)
print(a)
```

Dictionary

Read the docs[19]

Dictionaries are a collection of key value pairs, lists/tuples/sets are indexed sequences, dictionaries aren’t.

Here, ‘IN’ and ‘US’ are the keys and ‘India’, ‘United States’ are the values. The values can be any Python data type, but keys can only be hashable data types (basic data types, int/float/string/complex).

[17] https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
[18] https://docs.python.org/3/library/stdtypes.html#tuple
[19] https://docs.python.org/3/library/stdtypes.html#dict

```
i = {'IN':'India', 'US': 'United States'}
print(i['IN'])
print(i['US'])
```

Output: `India United States`

Set

Read the docs[20]

```
i = set([1,2,3,4,5])
```

Sets are same as lists, but they don't allow duplicates. Only hashable elements are allowed as their members (int, float, string, complex, tuples).

Immutability

When a data type is said to be immutable, the data type doesn't allow modifications.

Hashing

Hashing is a complicated process, just remember, hashable data types = immutable data types.

Links

|Next[21] | Previous[22] | Index[23] | —-| —-| —-|

[20] https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset
[21] 03-02-operators.md
[22] 02-more-about-language.md
[23] ../SUMMARY.md

Operators

Python has the following major operators:

`<, >, =, ==, >=, <=, !=, *, +, -, **, +=, -=, /=, *=, in, and, not, is.`

Operator priority

When expressions containing more than one operators are evaluated, the operator priority is followed, it is just like the BODMAS (PEMDAS for Americans) rule of maths. Usage of `()` can override priority.

```
>>> 3 * 2 + 20 - 46
-20
>>> 3 * (2 + 20 - 46)
-72
```

Assignment

= is the assignment operator.

```
>>> a = 1 # creates a new integer object of value 1
          # and stores the address in variable a.
>>> a = 2 # creates a new integer object of value 2
          # and stores the address in variable a.
>>> b = a # stores the address of variable a inside
           # variable b, they point to the same object.
```

Multiply

```
>>> a = 12
>>> a * 12
144
>>> a = "ab_"
>>> a * 2 # when * is used with strings,
          # it returns a new string twice.
ab_ab_
```

Exercise:

1. Take two numbers from the user and print their multiplication.
2. Take three numbers from the user and print their multiplication.

Add

```
>>> a = 1
>>> a + 1
2
>>> a = 'py'
>>> a + 'thon' # concatenates 'thon' to 'py' and creates a new string.
python
>>> a # we did not reassign a, so it's value is unchanged.
py
```

Exercise:

1. Take two strings from a variable and print their concatenation.
2. Take two numbers from the user and add them (you need to use the int() to convert the input to integer)
3. Take three numbers from the user and print their addition.

Equality

== is the equality operator, it returns true if both operands have the same value.

```
>>> a = 1 # creates variable a with value 1.
>>> b = 1 # creates variable a with value 1
>>> a == b # checks if a and b are equal.
True
```

> Note: It is a classic mistake to use == when you really want to use = or vice versa.

Division

`27/7` divides 27 by 7 and returns a floating point result

`27//7` divides 27 by 7 and returns an integer result.

```
>>> 27/7
3.85714
>>> 27//7
3
>>> 27%7
1
```

Power

`**` is the operator for calculating power.

```
>>> a = 2
>>> a**3
8
```

Shortcut operators

Consider that you have to create a variable `a = 3`. If you want to add `4` to the variable `a`.

You can do the following:

1. `a = a + 4`. But this tends to be verbose.
2. `a += 4`: Another way to do exactly the same calculation.

`+=` is a shortcut operator. There are other shortcut operators like: `+=, -=, /=, *=`. No spaces are allowed between `-=`, `+=`.

In other languages, you can use `++` or `--`, but they are not available in Python.

```
>>> a = 10
>>> a += 10
>>> a = a + 10 # same as a += 10
>>> a *= 10
>>> a /= 10
```

Membership test

The `in` operator tests if the element on the left hand side is present in the right hand side sequence (list, tuple, set).

```
>>> a = [1,2,3] # also works on set and tuples.
>>> 3 in a
True
```

Boolean operators

Read the docs[24]

not

Read the docs[25] `not` converts True to False and vice versa.

```
>>> not True
False
>>> not False
True
```

False like values

Variables of any data type when they are null or have no value, they are False like values. The negation of a False like value is True

[24]http://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not
[25]http://docs.python.org/3/library/stdtypes.html#truth-value-testing

```
>>> not '' # empty string is False like.
True
>>> not 0 # 0 is False like.
True
>>> not dict() # empty dict is False like.
True
>>> not list() # empty list is False like.
True
```

True like

Variables of any data type when have *some* value, any value, they are True like values. The negation of a True like value is False

```
>>> not 'dd'
False
>>> not 1 # non zero is True like.
False
>>> not -1 # non zero is True like.
False
>>> not [1,2,3] # list having any value is True like.
False
```

or

OR is true when either of the operand is true.

```
>>> True or False
True
>>> False or False
False
```

and

AND is true when both the operands are true.

```
>>> True and False
False
>>> False and False
False
>>> True and True
True
```

Comparision Operators

Read the docs[26]

There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, x < y <= z is equivalent to x < y and y <= z, except that y is evaluated only once (but in both cases y <= z is not evaluated at all if x < y is found to be false).

This table summarizes the comparison operations:

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Links

|Next[27] | Previous[28] | Index[29] | —-| —-| —-|

[26] https://docs.python.org/3/library/stdtypes.html#comparisons
[27] 04-list-set-dict.md
[28] 03-01-understanding-variables.md
[29] ../SUMMARY.md

High level data structures

In the last chapter we saw the various data types present in Python, in this chapter we'll look at the high level data types in detail.

List

Watch on YouTube[30] | Read the docs[31]

An ordered collection of n values (n >= 0)

Creating a list is simple:

```
>>> l = [] # creates an empty list.
>>> l = list() # creates an empty list.
>>> l = [1, 2, 3]
```

Each element of a list has an index, indices in Python start with `0`.

Let's say we have a list of five values, `a = [11,22,33,44,55]`.

Value	Positive Index
11	0
22	1
33	2
44	3
55	4

Indices can be negative.

Value	Negative Index
11	-5
22	-4
33	-3
44	-2
55	-1

List elements can be accessed using indices.

[30] https://www.youtube.com/watch?v=30S9LnvanwY
[31] https://docs.python.org/3/library/stdtypes.html#list

```
l = [11,22,33,44,55]
print(l[0]) # first value.
print(l[1]) # second value.
print(l[-1]) # last value.
print(l[-2]) # second last value.
print(l[-100]) # index out of range error.
```

List elements can be modified.

```
l[0] = 12 # will replace the value at the
         # index 0 to 12.
```

List elements can be used just like any other variable.

```
l[0]*12 # multiplies value at l[0] by 12.
```

List Methods.

append

The append function takes **one** argument and adds it to the end of the list.

```
>>> a = []
>>> a.append(1)
>>> a
[1]
>>> a.append("2")
>>> a
[1, '2']
# here '2' was added at the end of the existing list.
>>> a.append(1.11111)
>>> a
[1, '2', 1.11111]
>>> a.append([1,2,3])
>>> a
[1, '2', 1.11111, [1, 2, 3]]
# here, the entire list was inserted at the end of
# the existing list.
```

extend

```
>>> a
[1, '2', 1.11111, [1, 2, 3]]
>>> b = [1,2,3]
>>> a.extend(b)
>>> a
[1, '2', 1.11111, [1, 2, 3], 1, 2, 3]
# All elements of the list b were added individually
# to the list a.
```

pop

pop deletes and returns one element. It takes one optional argument.

- No argument is passed: Deletes and returns the last element.

```
>>> a
[1, '2', 1.11111, [1, 2, 3], 1, 2, 3]
>>> a.pop()
3
>>> a
[1, '2', 1.11111, [1, 2, 3], 1, 2]
```

- Valid index is passed: Deletes and returns the value at that index.

```
>>> a
[1, '2', 1.11111, [1, 2, 3], 1]
>>> a.pop(0)
1
>>> a
['2', 1.11111, [1, 2, 3], 1]
```

copy `a.copy()` creates a new list with the values of list `a`.

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> b = a # a and b are pointing to same object.
>>> a[1]=-111 # when we change a, b also changes.
>>> a
[1, -111, 3, 4, 5, 6]
>>> b # b changed when a changed.
[1, -111, 3, 4, 5, 6]
>>> c = a.copy() # creates a new list object.
                                 # having the values of list a.
>>> a
[1, -111, 3, 4, 5, 6]
>>> a[1]=999
>>> a
[1, 999, 3, 4, 5, 6]
>>> c # c didn't change when a changed.
[1, -111, 3, 4, 5, 6]
>>> b # b changed when a changed.
[1, 999, 3, 4, 5, 6]
```

Other functions Learning a language requires self practice! If we explain each and every function, we will hinder your path of exploring the language, we encourage you to use the `help()` function to find out more about other functions which are allowed on Lists.

Getting Help

We need to understand how to get help in Python.

- `help`: returns the documentation of the data type.
- `dir`: returns all the methods valid for that data type.

Example:

```
help(1) # help about integer class.
help('') # help about strings.
help(1.1) # help about float.
help([]) # help about lists
```

Output of `help`

```
>>> help('')
Help on class str in module builtins:

class str(object)
        |  str(object='') -> str
        |  str(bytes_or_buffer[, encoding[, errors]]) -> str
        |
        |  Create a new string object from the given object. If encoding or
        |  errors is specified, then the object must expose a data buffer
        |  that will be decoded using the given encoding and error handler.
        |  Otherwise, returns the result of object.__str__() (if defined)
        |  or repr(object).
```

Output of `dir`

```
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq_\
_', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass_\
_', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__'\
, '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'co\
unt', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isid\
entifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'p\
artition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swa\
pcase', 'title', 'translate', 'upper', 'zfill']
```

Slicing

Slicing is an easy way to fetch sub parts of them. Lists, strings and tuples allow Slicing.

The syntax of slicing is `l[start_index : end_index]`.

```
l: the list
start_index: Starting point with the element at this
             index included; defaults to 0 if left blank.
end_index: Ending point, but this index is excluded;
           defaults to -1 if left blank.
```

Slicing returns a new object (list or string or tuple) from the `start_index` to the `end_index` without including `end_index`.

```
>>> l = [0,1,2,3,4]
>>> l[0:3] # from 0 till 2nd element
           # the element of index 3 is not returned
[0, 1, 2]
>>> l[1:3] # new list starting from 1 till 2nd element
           # 3rd element is not returned
[1, 2]
>>> l[-1:] # new list containing the last element
[4]
>>> l[0:-1] # new list from 0 excluding last element
[0, 1, 2, 3]
>>> l[0:-2]
[0, 1, 2]
>>> l[::-1] # returns a new list with values reversed
[4, 3, 2, 1, 0]
>>> for i in l:
...     print(i)
...
0
1
2
3
4
```

Slicing also works with strings and tuples. We invite you to try them out. Don't just read this book, execute code!

Tuples

Read the docs[32] | Watch the video[33]

[32] https://docs.python.org/3/library/stdtypes.html#tuple
[33]

Tuples are read only lists. A tuple object, once created, doesn't allow us to add, delete or update an element. When we use the `dir` on a tuple object, we find that there are only two methods, `count` and `index`.

```
>>> a = (1,2,3,4)
>>> type(a)
<class 'tuple'>
>>> 1 in a
True
>>> a[0]
1
>>> a[::-1]
(4,3,2,1)
>>> a[0:2]
(1,2)
```

List vs Tuple

Lists are used when we are not sure how many values we'll be having.

Tuples are used when there is a fixed number of values to deal with.

Set

Watch on YouTube[34] | Read the docs[35]

Sets are same as lists with the following limitations:

1. Duplicate entries are not allowed
2. Sets can only have basic data types as elements (lists/dictionary/set/tuple)
3. Sets do not allow indexing

[34]https://www.youtube.com/watch?v=QmfDyjp0Z8E
[35]https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

```
>>> a = [1,2,3,4]
>>> b = set(a)
>>> b
{1, 2, 3, 4}
>>> type(b)
<class 'set'>
>>> b[0] # sets do not allow this.
TypeError: 'set' object does not support indexing
```

When we try to create a set from a list which has a list element, it is an error (only basic elements int, float, string are allowed) :

```
a = [1,2,3,[2,3]]
set(a)
## Throws an error.
```

Sets allow various methods like add, copy, deepcopy, update, pop, remove.

```
>>> a = set() # creates a blank set
>>> a.add("this")
>>> a.add("that")
>>> a.add("this and that")
>>> a
{'that', 'this and that', 'this'}
>>> a.remove("this")
>>> a
{'that', 'this and that'}
>>> a.pop()
'that'
>>> a
{'this and that'}
>>> a.add("zebra")
>>> a
{'zebra', 'this and that'}
>>> b = set([1,2,3,4,5])
>>> b
{1, 2, 3, 4, 5}
>>> b = set([1,2,3,4,5])
>>> c = b # c and b point to the **same** set object
>>> c
{1, 2, 3, 4, 5}
>>> b
```

```
{1, 2, 3, 4, 5}
>>> c.add(12) # changing c changes b
>>> b
{1, 2, 3, 4, 5, 12}
>>> c
{1, 2, 3, 4, 5, 12}
>>>
>>> d = b.copy() # creates a COPY of b
>>> d
{1, 2, 3, 4, 5, 12}
>>> b
{1, 2, 3, 4, 5, 12}
>>> b.add(1000) # changing b does NOT change d
>>> b
{1, 2, 3, 4, 5, 1000, 12}
>>> d
{1, 2, 3, 4, 5, 12}
```

Dictionary

Read the docs[36] | Watch on YouTube[37]

Dictionaries are key value pairs. Keys can only be hashable data types i.e. basic data types. There is no such restriction on the values.

Lists are indexed starting from `0`, it is done by the interpreter itself. Dictionaries allow custom indexes i.e. keys.

> Note: the representation of both set and dictionary is by using curly braces, `{}`.

```
>>> a = dict() # creates an empty dictionary
>>> a
{}
>>> a["IN"] = "India" # creates a new key value pair
>>> a
{'IN': 'India'}
>>> a["US"] = "United States of America" # new key value pair
>>> a
{'IN': 'India', 'US': 'United States of America'}
```

[36]https://docs.python.org/3/library/stdtypes.html#dict
[37]https://www.youtube.com/watch?v=pQV3wbSMBRI

```
>>> a["ES"] = "Espanol"
>>> a
{'IN': 'India', 'US': 'United States of America', 'ES': 'Espanol'}
>>> a["IN"] # returns the value of "IN" key.
'India'
```

Dictionaries do not support slicing.

`keys()` and `values()` are two functions which return all the keys and values of the dictionary object. Since we can't use `for i in` syntax to loop over dictionaries, we have to do this

```
>>> a.keys()
dict_keys(['IN', 'US', 'ES'])
>>> a.values()
dict_values(['India', 'United States of America', 'Espanol'])
```

```
>>> for i in a.keys():
...     print(i, ":", a[i])
...
IN : India
US : United States of America
ES : Espanol
```

We also encourage you to try out everything we did in this chapter, again! (on strings too)

Exercises:

1. Find the number of occurances of 1 in the list [1,2,1,2,1,2,3,4,1,2,3].
2. Find all the unique elements of a list. Input; a = [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4]. Output; [1, 2, 3, 4].
3. Solve the above problem using a dictionary.
4. Find the count of each element of a list except "\n". a = [1, 2, 2, 3] should return this output. 1 : 1,, 2 : 2, 3 : 1.
5. Create a random dictionary and print key : value pair in ascending order of keys. Input: a = {"IN":"India", "ES":"Espa⊠ol"}, output: "ES"":"Espa⊠ol", "IN":"India". You have to use `dir` to find out the necessary functions.
6. The same example as above, print in descending order.
7. Given two list, find their
    1. common elements.
    2. elements present in list a but not in b.
    3. elements present in list b but not in a. hint: use sets.
8. Print a reverse of a list without using the `reverse` method.

Links

|Next[38] | Previous[39] | Index[40] | —-| —-| —-|

[38] 05-constructs.md
[39] 03-02-operators.md
[40] ../SUMMARY.md

Constructs

del

`del` can be used to delete any variable, it de-allocates a variable. After you run `del a`, the variable `a` was deleted, so we can't access it.

```
>>> a
['2', 1.11111, [1, 2, 3], 1, 2, 3]
>>> del a
>>> a
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a
NameError: name 'a' is not defined
```

Exercise:

1. Given that a = [1,2,3,4,[5,6,7]]
    1. Delete the first value.
    2. Delete the second value of the list present at index 4.
    3. Delete the variable a.
    4. Take an input value from the user and delete it (if it is present), print "value not present" if the value is not present in the list.
2. Given that a = {"IN":"India", "ES":"Espanol"}
    1. Delete the key "IN".
    2. Delete the key "ES"
    3. Delete the variable a.

if-else statment

Read the docs[41] | Watch video 1[42]| Watch video 2[43]

Let's say we work for a school and we are asked to write a program to print out if a student has passed or failed for a particular exam. `a` is the score that the student scored in a subject, `b` is the maximum marks for the subject. Passing percentage is 35.

[41] https://docs.python.org/3/reference/compound_stmts.html#the-if-statement
[42] https://youtu.be/fbCsCFuj6zE
[43] https://youtu.be/YjUo6TQ2EzE

```python
a = 67
b = 100
percent = (a/b)*100 # calculating percentage marks.

if percent < 35:
    # block executes if percent is less than 35.
    print("Failed.")
else:
    # block executes if percent is greater than 35.
    print("Passed! Keep it up!")
```

Output:

```
Passed! Keep it up!
```

Indentation

Python uses spaces for indentation. Either spaces or tabs can be used for indentation, but not both. Usage of four spaces is recommended.

```python
a = 67
b = 100
percent = (a/b)*100 # calculating percentage marks.

if percent < 35:
    # block belongs to the if statement.
    print("Failed.")
    print("Please appear for exams again.")
else:
    # block belongs to the else statement.
    print("Passed!")
    print("Study for the next exams now.")
print("END")
```

Scoping

Each statement belongs to one or more "scopes", one direct scope and multiple indirect scopes.

For visualization, let's draw `[]` around the spaces, so the code now looks like this:

> Note: This is not valid Python code, `[]` is used just to show the indentation.

```
if percent < 35: # global scope
[    ]print("Failed.")
[    ]print("Please appear for exams again.")
else:
[    ]print("Passed!")
[    ]print("Study for the next exams now.")
print("END") # global scope
```

How to find scope?

1. After a construct which ends with a colon (if/for/while/try/except), calculate the number of spaces in the immediate next line. In this example, it is `four spaces`. The first `[]` block.
2. Every line below this line which has `four spaces` at the start until we get a line which **doesn't** have four spaces is in the if block.

Note: Having indentation **without** an if/for/while/elif block is a syntax error, for instance, you can't do the following:

```
if a > 1:
        print("hi")
print("bye")
        print("hi")
```

The error is because the interpreter isn't able to find out what block the statement belongs to.

elif

The `elif` block is special, it gets evaluated if the parent `if` block has been evaluated to False.

In our school, we now want to print the grades of students.

```
F: percentage < 35
C: 35 < percentage <= 50
B: 50 < percentage <= 70
A: 70 < percentage <= 90
A+ 90 < percentage
```

Note: `elif` and `else` can't exist without a corresponding `if` block. Only the `if` block can exist independently.

```
a = 67
b = 100
percent = (a/b)*100 # calculating percentage marks.

if percent < 35:
    # block belongs to the if statement.
    print("F")
elif percent <= 50:
    print("C")
elif percent <= 70:
    print("B")
elif percent <= 90:
    print("A")
else:
    print("A+")
```

Output: `B`

Nesting

We can have multiple if blocks inside an if block (same is true with for/while statements). This is called nesting. When we write nested if statements, then the same logic applies to indentation.

This is an example of two level nesting:

```
if a > 1:
        if b < 1:
                print(" b is less than 1")
        print("a is greater than 1")
print("This is not in either of the above blocks")
```

Let's add `[]` to the code to understand scoping for nested blocks.

Each `[ ]` can be considered as an indentation block, we can see that `if b < 1` has only one block, but this statement, `print(" b is less than 1")` has two blocks of spaces, the first one is to `if b < 1` which is the primary scope, the secondary scope is to `if a > 1`.

> Note: This is not valid Python code, `[]` is used just to show the indentation.

```
if a > 1:
[        ]if b < 1:
[        ][        ]print(" b is less than 1")
[        ]print("a is greater than 1")
print("This is not in either of the above blocks")
```

Scoping for nested blocks

Here, the statements which lie in the inner if statement lie in two blocks, the inner if statement's block and the outer if statement's block.

```
if True:
    # statements here are in the
    # scope of the if statement.
    print("Statement is true")
    if True:
        # statements here are in the inner if
        # but overall, lie in the scope of
        # the outer if.
        print("Another statement")
else:
    # statements here are in the
    # scope of the else statement.
    print("Statement is false")
    if True:
        # statements here are in the inner if
        # but overall, lie in the scope of
        # the outer else.
        print("Another statement")
```

Exercise

1. Take the name, age and score of the user as input. Print their percentage. If the percentage is less than 60, print F, if percentage is between 60 and 70 print B and if percentage is greater than 70 print A. Marks are out of 100.
2. Take a number from the user and print if it is even or odd.
3. Take two numbers from the user and print which number is largest.
4. Take two numbers from the user and print which number is smallest.
5. Take three numbers from the user and print largest and smallest number.
6. Take four numbers from the user and create a list.
7. Take four numbers from the user and create a set.

for loop.

Read the docs[44]

`for` statement is used to cycle over a list/set/tuple.

```
l = [1,2,3,4,5]
for i in l:
    print(i)
```

Output: `1 2 3 4 5`

Explanation:

The program goes through five iterations

Iteration 1: Value of i is 1; loop prints 1;

Iteration 2: Value of i is 2; loop prints 2;

Iteration 3: Value of i is 3; loop prints 3;

Iteration 4: Value of i is 4; loop prints 4;

Iteration 5: Value of i is 5; loop prints 5;

In a `for` loop, we can perform other operations too, for the sake of simplicity, we just printed the value of the variable i.

Another approach.

```
for i in range(len(l)):
    print(l[i])
```

Output: `1 2 3 4 5`

Explanation: `range(5)` returns a list [0,1,2,3,4].

The Else Block

`for` has an else block. It is strange at first glance, but it is quite helpful in certain cases, like finding if a number if prime or not.

We will take a simple example where we want to find out if an element is present in a list or not.

[44] https://docs.python.org/3/reference/compound_stmts.html#the-for-statement

```python
l = [1,2,3,4,5]
val = 99

for i in l:
    if i == val:
        # for understanding break
        # check the below section.
        break
else:
    print("Not present")
```

Output: `Not present`

Explanation: The loop is killed by using a `break`, if the loop does not get killed, then the value is not present in the list.

Exercise:

1. Print all numbers till 100.
2. Print all numbers from 20 to 100.
3. Print all even numbers less than 100.
4. Print all odd numbers less than 100.

break, continue, pass

break

Read the docs[45]

The break statement kills the current loop. If we have nested loops, then we have to position the `break` properly to kill the correct loop.

```python
for i in range(10):
    if i == 4:
        break
    print(i)
```

The above block will print all the numbers until it hits 4. The moment it hits 4, the for loop will stop executing.

continue

Read the docs[46]

The continue statement takes the execution to the next iteration.

[45] https://docs.python.org/3/reference/simple_stmts.html#break
[46] https://docs.python.org/3/reference/simple_stmts.html#continue

```
for i in range(100):
    if i == 4:
       continue
    print(i)
```

The above block will print all numbers **except** 4. It continues to the next iteration.

> Note: break and continue work with loops only, either `for` or `while`.

Exercise:

1. Print all numbers from 0 till 10 except 4 and 5.
2. Print all numbers from 0 till 20 except those divisible by 3.
3. a = [1,2,3,4,5], take an integer as input from the user. Print all numbers which are less than the input number using break.

pass

Pass can be used as an empty placeholder in places where you don't have anything to add.

```
if a > 1:
    pass
else:
    print("TODO")
```

For instance, in the above if-else block, you really don't know what logic you are going to put, so you can either use `print("TODO")`, or use the pass statement. pass doesn't print anything. You can use pass in any loop.

while loop.

Read the docs[47]

`while` is used when you have to loop for a specific condition. If you don't have a condition, you can use `True` and that would result in an infinite loop.

[47] https://docs.python.org/3/reference/compound_stmts.html#the-while-statement

```
i = 10
while i >= 0:
    print(i)
    i = i - 1
```

Output: `10 9 7 6 5 4 3 2 1 0`

Explanation: The `while` statement will loop on the condition, until the condition evaluates to `False`. The loop exits when the condition becomes False.

> Note: The `while` statement also has an else block, we encourage you to play with it to understand it better.

Exercise:

1. Print all numbers till 100 using while.
2. Print all numbers from 100 to 1 using while.
3. Print all even numbers less than 100 using while.
4. Print all odd numbers less than 100 using while.
5. For a list [1,2,3,4,5], write a program which checks if 6 is present in the list, using while.
6. Print all numbers from 0 to 10 except 4 and 5.

try - except - finally

Read the docs[48]

try-except is used for exception handling, we'll take a look at it in a later chapter.

with

> Note: requires the knowledge of file handling, you can come back to this chapter after reading the file handling chapter.

Read the docs[49]

The with block was added in PEP 343[50]. support of the Resource Acquisition Is Initialization[51] idiom commonly used in C++. It is intended to allow safe acquisition and release of operating system resources.

Resources are created within a scope/block, resources are cleanly released whether the block exists normally or because of an exception.

[48] https://docs.python.org/3/reference/compound_stmts.html#the-try-statement
[49] https://docs.python.org/3/reference/compound_stmts.html#the-with-statement
[50] https://www.python.org/dev/peps/pep-0343/
[51] http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

```
with open('file.py') as ip, open('op.txt','w') as op:
    for line in ip.readlines():
    op.write(line)
```

Output: `10 1 21 19 30 20`

Because of scoping, the `input_file` and `output_file` variables are only available within the with clause. This can result in some clean code, but it is upto you totally. I like to use try-except-finally.

We encourage you to read more about the `range` function. Please **do not use the Internet**, use the `help` function. The faster you get familiar to using the documentation, the better programmer you become. In a world full of people Googling "how to create a string in Python", we really need to be self sufficient as to using the documentation that a language provides to differentiate from others. The mark of a great programmer isn't in how much things she can store in her memory, it is in the mastery of the tools she uses, documentation and the help command are among the tools.

Links

|Next[52] | Previous[53] | Index[54] | —-| —-| —-|

[52] 06-file-handling.md
[53] 04-list-set-dict.md
[54] ../SUMMARY.md

File handling

Managing files is one of the most important features of any programming language. Python supports file handling via the io module's TextIOWrapper class, docs[55]

There are two basic modes in which we can manipulate files, Text and Binary.

Writing files

Write all even numbers till 100 to a file named "even.txt" and write all odd numbers to "odd.txt".

```
# open() opens a file and returns the address of the object
# To be able to write to a file, the second
# argument should be w.
even_file = open("even.txt", "w")
odd_file = open("odd.txt","w")

# Opening file in write mode
# Creates a file if it isn't present.
# If a file is present, it gets overwritten and data is lost.
# write mode doesn't allow the file to be read.

for i in range(100):
        if i % 2 == 0:
            # All IO operations happen via strings.
            # so argument of write should be passed a string.
            even_file.write(str(i))
            even_file.write("\n")
        else:
            odd_file.write(str(i))
            odd_file.write("\n")

# A file should always be closed
# to free system resources.
even_file.close()
odd_file.close()
```

[55] https://docs.python.org/3/library/io.html#io.TextIOWrapper

Why close a file?

When we write anything to a file, it gets written to a buffer and not directly to the underlying file. `flush()` method can be used to immediately write to the file. The `close()` method internally calls the `flush()` method, thus, in most of the cases, it is enough to just call the `close()` method.

Reading files

Read all the content from the file "even.txt" and print it to the terminal

```
# To be able to read a file, the second argument
# should be r.
input_file = open("even.txt", "r")

# Files which are already existing can be read.
# An exception is thrown if the file doesn't exist.
# Writing or appending the file is not allowed.

# readlines reads the complete file and returns a list
# of all lines.
lines = input_file.readlines()
for line in lines:
    print(line)
```

You will see that each number is printed on a different line. This is because of the new line ("\n") character which we wrote.

The case of the new line.

Whenever we read or write to a text file, we are responsible for the new line character "\n". When a file's content is "python\nis a good language.", it is displayed as

```
python
is a good language
```

on any text editor, because the "\n" gets converted into an actual new line. But, the content of the text file is still "python\nis a good language."

```python
file = open("file.txt", "w")
for i in range(5):
    file.write(str(i))
file.close()

file = open("file.txt", "r")
lines = file.readlines()
print(lines)
```

Output: [‘01234’]

We can see that the file contains only one line. This is because, the `write` method doesn’t add a new line character at the end of each operation. To understand it better, open this file in a text editor.

Now, consider this block.

```python
file = open("file.txt", "w")
for i in range(5):
    file.write(str(i) + "\n")
file.close()

file = open("file.txt", "r")
lines = file.readlines()
print(lines)
```

Output:

[‘0\n’, ‘1\n’, ‘2\n’, ‘3\n’, ‘4\n’]

By adding a “\n” in each write call, we ensure that the new number gets written to a new line, that’s why, when we run the above code, we will get four lines in the text file. Verify this by opening the file in a text editor.

While reading a file, we need to deal with the “\n” present at the end of each line. There is a shortcut to remove the “\n”.

```python
f = open("even.txt")
lines = f.readlines()
print(lines)
lines = [line.strip() for line in lines] # list comprehension
print(lines)
```

List comprehension

It is a shortcut of working with lists, to perform filter or some other operation on the entire list.

```
lines = [line.strip() for line in lines]
```

List comprehension replaces the following block:

```
for i in range(len(lines)):
    lines[i] = lines[i].strip()
```

Syntax

```
[ condition for i in <list> if <another condition>]
```

List comprehension returns a new list based on the current list, the first argument `list.strip()` would be the elements of the list. The second argument is the for loop, the third argument is optional, you can have an `if` block there.

```
a = ["a.txt", "b.txt", "c.tct", "j.txt"]
# list comprehension to remove *.tct
b = [i for i in a if not a.endswith(".tct")]
# write another list comprehension to remove all .tct files
```

Examples:

1. Convert each element to upper case (works only when each element is a string). `python a = ["a.txt", "b.txt", "c.tct", "j.txt"] b = [i.upper() for i in a] print(b)`
2. Convert each element to lower case (works only when each element is a string). `python a = ['A.TXT', 'B.TXT', 'C.TCT', 'J.TXT'] b = [i.lower() for i in a] print(b)`
3. Give all the values of list a which are greater than 3.

```
a = [1, 3, 4, 5, 5]
b = [i for i in a if i > 3]
print(b)
```

4. Give all the values of list a which are greater than 3. `python a = ["Haskell", "Ruby", "Python"] b = [i for i in a if len(i) > 3] print(b)`
5. Give all values in a which are even.

```
a = [1, 3, 4, 5, 5]
b = [i for i in a if i % 2 == 0]
print(b)
```

Appending files

Write even numbers from 100 to 200 in a file “even.txt” which already contains even numbers from 0 to 100.

```python
# To append to a file, the second argument should be a.
# if we open in write mode, we will lose the data present
# in the file before we open it.
file = open("even.txt", "a")

# Does not overwrite the file like the write mode.
# It will create a file if it isn't present.
# Adds content to the end of the file.
# Doesn't allow file to be read.

for i in range(100, 200):
        if i % 2 == 0:
                file.write(str(i)+"\n")

file.close()
```

Binary Mode

```python
f = open("file.txt", "rb")
```

Python allows us to manipulate binary files, we have to club the modes, `b` stands for binary, along with `b`, we can use any of r/w/a.

```
* Read a binary file: "rb".
* Write a binary file: "wb".
```

> Note: One essential part of working with files is removing the `\n` or adding it when it is required.

Other methods

1. `read()` Takes an argument as number of bytes to be read.

   When you open a file for reading, the pointer is at the 0'th position. if you do `f.read(1)`, the pointer moves to 1. If you do `f.read(1)` again, the pointer would return the 2nd character of the file and not the first one.

   All the read functions returns characters based from the current pointer, you can know where the pointer is located currently by using `f.tell()`.

   > Note: This is a session for the Interpreter.

```
>>> f = open("lines.txt")
>>> f.tell() # pointer is at 0th position
0
>>> f.read(1) # returns the first character
'f'
>>> f.tell() # pointer shifted to 1st character
1
>>> f.read(20) # reads 20 characters after 1 ('\n' is considered as single characte\
r)
'irst line\n second li'
>>> f.tell()
21
>>> f.read(10000) # reads 17 char after 21
'ne \n third line \n'
>>> f.tell() # pointer on 38
38
>>> f.read(10) # read 10 more char
''
>>> f.tell() # this is the last position!
38
>>> f.seek(0) # pointer is now at start of file
0
>>> f.read(1)
'f'
```

In this example, you hit the end of the file, there are only 38 characters in it, "\n" also is a character. If you read beyond the max characters of a line, it will return you blank strings. but there might be cases in which you would want to start reading the file again from any other position.

You can use `seek` function to reset the pointer at any position of your choice. Let's say that you want to re-read the file again

```
>>> f.seek(0)
```

This will reset the pointer and you should be able to read it from the first character of the file!

2. `readline()` Returns the current line that the pointer is located.

```
>>> f.seek(0)
0
>>> f.readline()
'first line\n'
>>> f.readline()
' second line \n'
>>> f.readline()
' third line \n'
```

You can try doing `f.seek(0)` and reading the lines again, you'll get the same output as above.

3. `readlines()` Returns all the lines of the file as a list.

   If the pointer is present at the end of file, then it returns an empty list. In that case, do a `f.seek(0)` to reset the pointer.

```
>>> f.seek(13)
13
>>> f.readlines()
['econd line \n', ' third line \n']
>>> f.seek(0)
0
>>> f.readlines()
['first line\n', ' second line \n', ' third line \n']
```

4. `write()` Writes the **string** argument which you pass on the pointer. We need to be careful of the pointer, otherwise writing to the file can be dangerous. You are responsible for adding the "\n" character if you want to a new line, by default it'll write at the position the pointer is currentl `f.tell()`.
5. `writelines()` If you want to write a list to a file, you'll use writelines. Even here you are responsible for adding \n wherever you feel necessary.

```
j = ["\nthis is something\n", "another line\n"]
f.writelines(j)
f.close()
```

We encourage you to try out file IO on your own, please refer to help() for any details regarding file, do not search "how to open a file in Python" on the internet.

Exercise

1. Take the user's name, age and height as input and write them to a file using this format. name,age,height.
2. Read a file "input.txt" and print all the lines which are present multiple times. Please create the input.txt file in such a way that it has many lines and few of them are present multiple times.

3. Read a file “input.txt” and print how many times each line is present.
4. There are two files, “file1.txt” and “file2.txt”
    1. swap the content of both files.
    2. append the content of file1.txt to file2.txt.
    3. append content of file2.txt to file1.txt.
    4. take unique content of both files and write them to file3.txt
5. Use any file created above. Take a positive number as input from the user and read those many characters from the text file.
6. Prepare a csv file like this: first field = name, second onwards marks/100.

```
tom,10,10,10
tim,20,20,20
```

1. Print the name of the student with highest marks.
2. Print the name of the student with highest marks in upper case.
3. Print the highest score.
4. calculate the total score of each student, add the total at the end of the line and write everything to result.csv

sample output:

```
tom,10,10,10,30
tim,20,20,20,60
```

7. Read the result.csv file and create a new file named result.txt like the following sample output.

```
tom-10-10-10-30
tim-20-20-20-60
```

8. read the result.txt file. Print every other character from the file.

Links

|Next[56] | Previous[57] | Index[58] | —-| —-| —-|

[56] 08-exception.md
[57] 05-constructs.md
[58] ../SUMMARY.md

Exception handling

When something goes wrong, Python throws an an exception object. `Exception` is a class from which other exceptions are derived. The name of the child classes usually end with Error like FileNotFoundError.

Read the docs[59]

file: exception.py

```
i = 0
j = 100/i
print("This is the print statement")
```

This code won't run, it would throw the ZeroDivisionError exception.

The problem with exceptions is that your program is killed the moment it throws an exception. When writing large scale software in Python, this becomes an issue. This is where exception handling comes into picture. We want our programs to be resilient about errors that might happen during the runtime and we do not want our code to exit for any such reasons.

file: exception2.py

```
i = 0
try:
    j = 100/i
except:
    print("Can't divide by 0!!")
```

When you run this program, you'll see that it doesn't kill the program.

A try catch block is to be used for lines which we suspect that something might go wrong, for instance if you are trying to open a file, there are many things which could go wrong, for instance, the disk space would go full, the file might not exist or a million other things, thus, we wrap the `open` function call in a try catch block.

The `try` block contains all the lines where we suspect that something might go wrong, the `except` block is where the damage control lines are present. What we do for handling exceptions is upto us, we can just print on the terminal or we can send emails to the respective teams, or just log that there was an exception.

[59] https://docs.python.org/3/tutorial/errors.html#errors-and-exceptions

```
i = 0
try:
    j = 100/i
except ZeroDivisionError as e:
    print(e)
except Exception as e:
    print(e)
finally:
    print("this block always gets executed")
```

The try-catch block allows you to handle multiple exceptions, they must be in the reverse order of generality. For instance, if you want to handle every kind of exception, just use ‘ except Exception‘, the smallest exception statement should be at the top.

Finally

The finally block gets executed every time a try block gets executed. You can put the statements in this block which you want to be executed every time.

You’d be doing this when you do handle errors for file handling programs.

```
try:
        f = open("lines.txt", "r")
except Exception as e:
        print(e)
```

Of course, you can pick up something other than the Exception class. If you want to handle some specific scenario like FileNotFoundError, use it instead of the Exception as e class. When you use `Exception as e`, you get the exception thrown by the code into an object `e`, you can print the object or write it to a file. If you don’t care for that, just use `except Exception:`, skip the `as e`.

Links

|Next[60] | Previous[61] | Index[62] | —-| —-| —-|

[60] 09-functions.md
[61] 07-examples.md
[62] ../SUMMARY.md

Functions

Functions are used to save code duplication. When we want a block of code duplicated across multiple locations, rather than physically copy pasting the code, it is better to declare a function and use it wherever we require it.By doing this, we write easy to edit code, because when our logic changes, we have to make changes only to the function definition and not everywhere the block is present.

file: func.py

The following is the typical function definition on Python, def is the keyword. 'function_name' is the name to be given to the function.

Functions support zero or more arguments.

```
def function_name(argument1, argument2):
    <statements>
```

Let's write a simple program to print the square of a number passed to it as an argument.

Invoking functions.

```
def square(number1):
    print(number1**2)
```

Save the file and run it.

You will notice that the program didn't give any output, and that is because we just defined a function. When the interpreter comes across the def block, it will create a function of that name, taking some arguments and performing some action.

It will not execute the function, if we want the function to be used, we need to invoke it.

Add `square(2)` to the end of the above file. Save and run the program again. This time, you'll see the output. This is called function invocation.

Here, 2 is an argument which is stored in number1 during the function invocation. Just because an argument is defined doesn't mean that we have to use it, we can ignore it altogether if we want.

Default arguments

Now that we wrote the square function, we want to generalize it to calculate power. This function will take two arguments, number and the power, and return number ** exponential.

file: defaultargs.py

```
def power(number, exp=2):
    print(number ** exp)

power(2, 3) # value of exp is 3.
power(2) # since no value of exp
         # is given, defaults to 2.
```

Some or all arguments in Python can be optional. When an argument is optional, the default value of the argument is used.

The rule with default arguments is that they need to be at the right most side of the argument listing, after all the mandatory arguments.

The following block of code will result in a SyntaxError. "'python def do(var1=2,var2): print("hi")

do(1) "'

Return

In the above examples, we saw function which prints the output. A function can also return the its output to the caller. This is helpful when processing data which we do not want to print.

There is no restriction on the number of values that can be returned. Values are returned using the return statement and it is not a part of the function signature, a return statement should be directly added in the function definition.

```
def add(name, mode):
    return name+mode, name-mode

one, two = add('this', 'that')
print(one, two)
```

Here, we have a function which returns two values. The two values are then printed after the function call.

Links

|Next[63] | Previous[64] | Index[65] | —-| —-| —-|

[63] 10-task.md
[64] 08-exception.md
[65] ../SUMMARY.md

Building a todo list manager

From this chapter on, we will start building an application. The end goal is going to write a command line todo list manager.

The functionality is going to be something like this:

```
$ python tasks.py add "title of the task" "content of the task"
new task added

$ python task.py remove "title of the task"
task deleted

$ python task.py list
task_one content_of_1
task_two content_of_2
```

The software design phase is the most critical phase of software development, writing software is a cakewalk when the design is done properly.

Before we can write the code, we have to:

1. Identify the input source. Our input source will be command line arguments.
2. Define the functionality of the program. Program will support adding/removing/listing todo items.
3. Identify where to store data. Program will store content in a flat file.
4. Identify what will be the output. Program will print data on the terminal.

Input

The user is going to give input in the command line arguments.

The ‘sys’ package has the command line arguments stored into a variable called ‘argv’.

file: tasks1.py

```
import sys
print(sys.argv)
```

Try running the code, you will see something like this

```
ch10 $  python3 tasks.py
['tasks.py']

ch10 $  python3 tasks.py "title" "content"
['tasks.py', 'title', 'content']
```

For the interpreter, tasks.py, title and content are all command line arguments, this is what you see as the output of the above script.

Output

The output of the program is going to be on the command line.

Formatting output

file: tasks2.py

```
import sys
for i, item in enumerate(sys.argv):
    print("{0} {1}".format(i, item))
```

Enumerate takes a sequence and returns (index,value) for each value.

Formatting.

Starting from Python3.6, we can do this:

```
print(f"{i} {sys.argv[i]}")
```

Where, `i` and `sys.argv[i]` are variables. Read the docs[66]

Try running tasks2.py with various command line arguments.

[66]https://docs.python.org/3/whatsnew/3.6.html#pep-498-formatted-string-literals

```
ch10 $ python tasks2.py
0 tasks2.py
```

We didn't give any parameter, since we know that the 0'th argument is going to be the file name itself, there are no suprises here.

```
ch10 $ python tasks2.py title content
0 tasks2.py
1 title
2 content
```

We passed title and content, and the index 1 is title, index 2 is content.

```
ch10 $ python tasks2.py add  title content
0 tasks2.py
1 add
2 title
3 content
```

We passed "add title content" as the command line argument, and we get the expected output.

Adding commands

file: tasks3.py

```
import sys
args = sys.argv

command = args[1]

if command not in ("add","remove","list"):
        print("Invalid command, Use add/remove/list")

if command == "add":
        print("adding")
elif command == "remove":
        print("removing")
elif command == "list":
        print("listing")
else:
        print("invalid command!")
```

You'll get this output when you run the code, this is an `exception`.

```
ch10 $ python3 tasks3.py
Traceback (most recent call last):
File "tasks3.py", line 5, in <module>
        command = args[1]
IndexError: list index out of range
```

What went wrong?

We encourage you to try to figure out what went wrong, before moving ahead.

Handling errors

In the earlier file, we tried to access an list index which didn't exist. This raised an exception, we saw in the chapter about exceptions that a try-catch block is used to handle exceptions.

Update the `command = args[1]` line in the file to this block: `python try: command = args[1] except IndexError: print("Invalid arguments!") sys.exit(1)`

Exit

`exit` kills the program execution with the ID of what we pass in as an argument, `0` is successful exit, anything greater than `0` is unsuccessful exit

We catch IndexError exception. If there is an IndexError exception that means that the user has not given the appropriate arguments.

Save and run the file. The output should be this:

```
ch10 $ python tasks3.py
Invalid arguments!
```

We have handled the scenario where the user gives less input than what is required. Now let's move ahead and type an invalid command.

```
ch10 $ python tasks3.py random
Invalid command
Use add/remove/list
invalid command!
```

We can see that the "invalid command" message is being repeated twice, we have to do something about that.

We add another exit call after printing Invalid command. The reason being there is no need to go any further when we have established that the user has given us the invalid command.

```
if command not in ("add","remove","list"):
        print("Invalid command\n Use add/remove/list")
        sys.exit(1)
```

```
ch10 $ python tasks3.py random
Invalid command
Use add/remove/list
```

Now, let's test the `list` command.

```
ch10 $ python tasks3.py list
listing
```

Storing user data

Now that we have finished getting started with our menu driven program, let's go ahead and create a list. We need a variable to store the task list. When the program would be used the additions and deletions would be done on this list object, which would be written to the file when the output is required.

Add this line after `args = sys.argv`

```
tasks = []
```

This will create a variable by the name `tasks` which is visible in this file to all functions.

In the `list` block, we want to now print the values stored inside `tasks` variable. If the values aren't present, we should print "No tasks present", if there are tasks, then we should print the elements inside `tasks`.

We have to use the `len()` function to check if there is nothing in the variable.

Update this block.

Listing tasks

```python
# This is a snippet
# can't have elif without parent if
elif command == "list":
        if len(tasks) == 0:
                print("there are no tasks!")
        else:
                for task in tasks:
                        print(task)
```

We now simulate data, before we let the user have the ability to add a task, we will populate the task variable by ourselves.

For simplicity, we choose this format, the title and content would be concatenated by a | character.

update the `tasks = []` to this line, `tasks = ["title|content"]`.

And the else block of len(tasks) to this

```python
for task in tasks:
        title, content = task.split('|')
        print("{0} {1}".format(title, content))
```

Adding a task

We will now work on adding a new task. The input would be taken from the command line argument.

```python
if command == "add":
        print("adding")
```

This block is changed to:

```python
if command == "add":
        title = args[2]
        content = args[3]
        task = title + content
        tasks.append(task)
```

But changing this does nothing, this is because the `tasks` variable is stored during the runtime. It gets reset to the default variable when the program quits. We need to add file handling feature to store the task list.

Replace the `tasks` line to this to store an empty variable.

```
tasks = []
```

The if-else block should look like this:

```
if command == "add":
        title = args[2]
        content = args[3]
        task = title + content
        file = open("tasks.txt", "a")
        file.write(task+"\n")
        file.close()
elif command == "remove":
        print("removing")
elif command == "list":
        file = open("tasks.txt", "r")
        tasks = file.readlines()
        if len(tasks) == 0:
                print("there are no tasks!")
        else:
                for task in tasks:
                        title, content = task.split('|')
                        print("{0} {1}".format(title, content))
        file.close()
```

Our first bug!

If you run this file, you'll get an IOError saying that tasks.txt doesn't exist. This is because we have not handled this scenario in the `open` function. We need to wrap that in a try-except block.

```
ch10 $ python tasks3.py list
Traceback (most recent call last):
File "tasks3.py", line 28, in <module>
        file = open("tasks.txt", "r")
IOError: [Errno 2] No such file or directory: 'tasks.txt'
```

In the elif block of list, we make the following modifications:

```
try:
        file = open("tasks.txt", "r")
except IOError as e:
        print(str(e))
        sys.exit(1)
tasks = file.readlines()
```

Now when we run the code,

```
ch10 $ python tasks3.py list
[Errno 2] No such file or directory: 'tasks.txt'
```

This is a graceful handling of the scenario where we aren't able to access the file due to an I/O (Input/Output) operation error.

```
ch10 $ python tasks3.py add "new task" "new content"
```

Now, let's list the tasks.

```
ch10 $ python tasks3.py list
Traceback (most recent call last):
File "tasks3.py", line 38, in <module>
        title, content = task.split('|')
ValueError: need more than 1 value to unpack
```

Now, we run the add command and try to list the values. We get an error, we can't add a try-except block to everything, so it is necessary to figure out what the issue is. Here, when we do `cat tasks.txt`, we come to know that the content of the file is this.

```
ch10 $ cat tasks.txt
new tasknew content
```

We don't have a | character between the title and content! We did a mistake when we concatenated title and content. Remove the file by doing `rm tasks.txt`, or, delete the file manually if you are on windows.

Instead of `task = title + content`, we need this, `task = title + "|" + content`.

This is the output now

```
ch10 $ python tasks3.py add "new title" "new content"
ch10 $ python tasks3.py list
new title new content
```

This is great! We now are able to add and list the tasks.

Note:

When giving input over the command line, if you want to give multi word input, please enclose them in either single or double quote. For instance, we gave the input “new title”, because our title contained a space. If we had given tasks2.py add new title, “new” would be considered the title because space is the delimiting character for any command line input, hence the “new title” enclosed in quotes.

```
ch10 $ python tasks3.py add "Finish Python book" "Working on 10'th chapter"
ch10 $ python tasks3.py list
new title new content

Finish Python book Working on 10'th chapter
```

You can see that the output of the list command isn’t particularly good, so let’s use the advanced features of the print function for this.

Replace the else block of `if len(tasks)==0` by this.

```
print("|-----{0}----{1}----|".format("title", "content"))
tasks = [task.strip() for task in tasks]
for task in tasks:
        title, content = task.split('|')
        print("|-{0}----{1}-|".format(title, content))
```

Format specifiers enable us to control the layout of the print, we encourage you to try various things out.

The final code should look like this.

```
import sys

args = sys.argv

tasks = []

try:
        command = args[1]
except IndexError:
        print("Invalid arguments!")
        sys.exit(1)

if command not in ("add","remove","list"):
        print("Invalid command\n Use add/remove/list")
        sys.exit(1)


if command == "add":
        title = args[2]
        content = args[3]
        task = title + "|" + content
        file = open("tasks.txt", "a")
        file.write(task+"\n")
        file.close()
elif command == "remove":
        print("removing")
elif command == "list":
        try:
                file = open("tasks.txt", "r")
        except IOError as e:
                print(str(e))
                sys.exit(1)
        tasks = file.readlines()
        if len(tasks) == 0:
                print("there are no tasks!")
        else:
                print("|-----{0}----{1}----|".format("title", "content"))
                tasks = [task.strip() for task in tasks]
                for task in tasks:
                        title, content = task.split('|')
                        print("|-{0}----{1}-|".format(title, content))
        file.close()
else:
```

```
        print("invalid command!")
```

Removing tasks

To remove tasks, we have to change the way we structure our data. We either can accept deletion on the basis of the title of the task, or we can render index for each task, since deletion from the title is not exactly scalable (two tasks can have the same title but different content), we choose to modify our program to show index for each task, that way, the user can just give the index of the task which they want to delete.

file: tasks4.py

We first need to modify the way we represent our tasks to the user, instead of showing just the title and content, we will show the index too. For this, we need to make the following changes.

We can't loop like `for task in tasks`, we need to loop using `range`, `for i in range(len(tasks))` is the way to go. The only difference is that we have to then fetch the task as `tasks[i]` rather than just `task`, because now, there is no such variable as `task`.

```
## Snippet, else can't exist without parent if
else:
        print("|-{0}----{1}----{2}----|"%("index", "title", "content"))
        tasks = [task.strip() for task in tasks]
        for i in range(len(tasks)):
                title, content = tasks[i].split('|')
                print("|-{0}--{1}----{2}-|" %(i, title, content))
```

In the actual delete block, we will use the del keyword which will simplify our task greatly.

```
# Snippet
elif command == "remove":
        task_id = args[2]
        del tasks[task_id]
```

```
ch10 python3 tasks4.py remove 0
Traceback (most recent call last):
  File "tasks4.py", line 27, in <module>
    del tasks[task_id]
TypeError: list indices must be integers or slices, not str
```

Note:

We are not validating if the user has given appropriate input, let's say the user gives `python tasks4.py remove` instead of `python tasks4.py remove 0`, then our program should complain about an error, the same is the case with add, if the user doesn't give both title and content, that's an error and it should be handled appropriately.

We can see that "list indices must be integers" is the error we got for the del statement, the reason for that is that as we said, all shell operations are string based, so when the user gave us the input `0`, it was '`0`', thus a string. We will typecast the `task_id` variable to an integer. Change it to this below statement.

We also need to read the file, for each instance, we read the file or appended it as required.

```
del tasks[int(task_id)]
```

We also need to write the updated `tasks` variable to our file, we add a "\n" to each element using the list comprehension mechanism.

Now try running the code. "'python ## snippet elif command == "remove": try: file = open("tasks.txt", "r") except IOError as e: print(str(e)) sys.exit(1)

```
file.close()
tasks = file.readlines()
tasks = [task.strip() for task in tasks]
task_id = args[2]
del tasks[int(task_id)]

file = open("tasks.txt", "w")
tasks = [task + "\n" for task in tasks]
file.writelines(tasks)
```

```
Output:

        ch10 $  python3 tasks4.py remove 0
        ch10 $  python3 tasks4.py remove 1

###### Note:
We do not print confirmation like "task deleted", "task added", but you can add them\
 if you want.

We have a fully working todo list manager as of now, what we need to do, is to reduc\
e the redundancy. That'll be undertaken in the next chapter.
```

```
## Homework
Translate this todo list app to use sqlite3 database which comes inbuilt with Python\
. Please do not use the Internet, use the sqlite3 documentation.

##### Links

|[Next](11-function-tasks.md) | [Previous](09-functions.md) |  [Index](../SUMMARY.md)
| ----| ----| ----|

{bump-link-number}

{leanpub-filename="11-function-tasks.md"}

# Adding functions

> Note: Please see the accompanying code. https://github.com/thewhitetulip/code-buil\
d-app-with-python-antitextbook

In the earlier chapter we saw how to take input, delete a task, but if you see the c\
ode, `task4.py`, there is a lot of redundant code, what we need to do is write funct\
ions to save repeated code.

We first start by defining the `main` function. There is no special significance to \
the main function, just that we chose to call it as `main`, we can very well choose \
to call it 'somerandomfunctionasfasdf'.

We push everything _except_ the import statement into the main function.

If you try to run the file at this point, there would not be any output, the reason \
for that is that you have declared a function but not **called** it. The interpreter\
 runs the script and it creates a function named "main" and it does nothing. If you \
want to **run** the main, you have to call it. At the bottom, add `main()` and then \
try running the file, this time, it'll give _some_ output.

Now, we create three functions, `add_task`, `remove_task`, and `list_task`. These th\
ree would do the respective functions.

The logic of our program should be split between modules. There should be one "contr\
oller" module which handles the IO and a supporting library which does something wit\
h the data, in that way, we can enable different input sources for the same app. Cur\
rently, the input source, which is the command line arguments is deeply coupled with\
 our todo list manager, so if tomorrow, we want to take input from some other source\
, we have to rewrite the complete application.
```

````
We manage the functions in this way:

main: handles the IO

add_task(title,content): adds a new task with title and content

list_task(): lists the tasks

remove_task(index): deletes the task of index.

By doing this, the main if-else ladder looks like this

```python
if command == "add":
    title = args[2]
    content = args[3]
    add_task(title, content)
elif command == "remove":
    task_id = args[2]
    remove_task(task_id)
elif command == "list":
    list_task()
else:
    print("invalid command!")
````

Now, if we change the input from command line to say FTP, all we have to do is change the main function, write the FTP input function and call add_task, with the newly fetched title and content. Modularity is really important in programming.

We have no need of declaring the `tasks` variable in the main function, so we remove it.

We also do not want to write "add" every time we refer to the add command, so we define three variables.

```
ADD,REMOVE,LIST = "add","remove","list"
```

So, the next time we want to refer to "add", we will refer to ADD.

This block also changes.
```

```
    if command not in (ADD,REMOVE,LIST):
        print("Invalid command\n Use {0}/{1}/{2}".format(ADD,REMOVE,LIST))
        sys.exit(1)
```

We now try and run the code, and it works!

```
ch10 $ python3 tasks4.py
Invalid arguments!
ch10 $ python3 tasks4.py list
|-index----title----content----|
|-0--new title----new content-|
|-1--Finish Python book----Working on 10'th chapter-|
```

The code looks cleaner than tasks4.py, but there is work to do! We can create a module for these three functions and reuse the file object instead of redefining the object in each function.

A parting glance at the design of the functions. Each function should do *one* thing and do it well. That way, our program is split into multiple functions which do one thing well, just like the UNIX philosophy.

Links

|Next[67] | Previous[68] | Index[69] | —-| —-| —-|

[67] 12-modules-tasks.md
[68] 10-task.md
[69] ../SUMMARY.md

Using modules

> Note: Please see the accompanying code. https://github.com/thewhitetulip/code-build-app-with-python-antitextbook

In the last chapter, we left off by creating three functions for our core functionality. In this chapter, we'd split the functions into two packages,

1. the main package
2. the todo package (which we'll create)

The `todo` package is going to contain everything related to our todo manager and the main package is just going to be the executable program.

The first thing we do, is create a `main.py` file, and cut paste our main function inside it and remove the invocation at the bottom of the `tasks.py` file.

So now our code is split into two files, in Python3 a file can be imported as a package directly, it should just be present in the same directory.

```
import tasks as t
```

By doing this, we can access all the members of the `tasks.py` file by appending 't.', like `t.list_task()`.

Replace the three functions according to this rule. If we had not given the alias t to the library, as is the keyword used to give alias, then we'd have to write `tasks.list_task()`, but since we are lazy, we prefer to give it an alias to call it like `t.list_task()`.

Modules are one of the most important aspects of Python. When we import a module, the interpreter **executes** the file which points to module.py (if we import a module named module).

This is the way any module is imported:

1. The interpreter first goes to the stdlib folder and tries to find the .py file
2. If found, it executes it
3. If not found, it looks in the current directory
4. If found, it executes it
5. If not found, it complains as ModuleNotFoundError: No module named ""

Special care needs to be taken so that we don't create a file with the same file name as a stdlib package, the humans reading the code would surely get confused, even if Python won't!

As an example, try creating a file `sys.py` in the current folder. Add one line to that file, `print("something")`. Add an import statement `import sys` and later run `python3 tasks.py`, you'll see that it doesn't print "something", it refers to the stdlib package called 'sys'.

Make Code Great Again!

There are many ways that we can make our code great again (ahem), there are lots of features we can add, a partial list can be as below:

1. add a "task status" field
2. change the formatting to show a checkbox like

   [ ] Do this and that (for incomplete)

   [x] Do this and that (for complete)
3. ability to search
4. ability to set deadline
5. log the time of operations
6. show graphs based on task completion.

But since this is not a perfect world, we'd not be doing all that, in this tutorial we hoped to teach you the way to develop an app, so you can build everything any feature you want, make sure that you ping me if you upload it to Github!

Now, we will make a few changes to our tasks library. We do not want the open() call three times, so we will wrap it in an function of our own, that'll save us multiple open calls.

```
def open_file(name, mode):
    try:
        file = open(name, mode)
    except IOError as e:
        print(str(e))
        sys.exit(1)
    return file
```

This will be our function which wraps the file open statement for now.

Why would we need such a thing? Why would we bother to wrap the open in an abstract function? The reason behind this is that by doing so, we have migrated the entire try-catch block inside one function. Now, if we were to change something there, we just have to change one block. Hence, the UNIX philosophy, one program does one thing well.

The case of the `__name__`

Since packages are executed as just another python program when they are imported, there is this case that we need to handle.

At the end of tasks.py, add a print statement.

```
print("this is some print statement")
```

Save the file and run the main file.

```
ch12 $ python3 main.py
this is some print statement
Invalid arguments!
```

You can see that the print statement got executed, as we said, a module when imported is executed like just another python program. The solution to this problem is to add an if block.

```
if __name__ == "__main__":
    print("this is some print statement")

ch12 $ python3 main.py
Invalid arguments!
```

Now we can see that the output is the way we want it. The `__name__` is the variable which stores the way the file was executed, if we run it like `python3 file.py` then `__name__` stores the value ‘**main**’, otherwise, it stores how it was invoked, in this case, it was invoked as “import tasks”, thus `__name__` stores the value tasks.

This example of course was basic, but when we write test cases for the functions of that module, they need to be inside this if block, so that we don’t accidentally run the entire suite of test cases whenever we import the package.

To check the value of `__name__`, add

```
print(__name__)

if __name__ == "__main__":
    print("this is some print statement")
```

At the bottom of `tasks.py` NOT in the if block, if you add it to the if block then it won’t be executed unless the file was executed.

Links

|Next[70] | Previous[71] | Index[72] | —-| —-| —-|

[70] 13-examples.md
[71] 11-function-tasks.md
[72] ../SUMMARY.md

Some examples

1. Merge two files
2. Student Marks manager
3. CSV to SQL generator

Merge two files

Let's say you have two files, "file.txt" and "file2.txt" and you want to merge them. One can try sorting the file and then copy pasting and running some duplicate removal program, or one can use Python.

First read both the files and create a dictionary. In that dictionary, we create an integer entry about each line, if it is not present in the dictionary keys, we add a value, otherwise we increment the value, that way, we can find out the duplicate values if needed.

file name: ch13/merge.py

```
first_file = open("file.txt")
second_file = open("file2.txt")

first_lines = first_file.readlines()
second_lines = second_file.readlines()

first_lines = [line.strip() for line in first_lines]
second_lines = [line.strip() for line in second_lines]

final_lines = {}

for line in first_lines:
    if line not in final_lines.keys():
        final_lines[line] = 0
    else:
        final_lines[line]+=1

for line in second_lines:
    if line not in final_lines.keys():
        final_lines[line] = 0
    else:
        final_lines[line]+=1

lines = "\n".join(list(final_lines.keys()))
```

```

file = open("output.txt", "w")
file.write(lines+"\n")
file.close()
```

Student Marks Manager

Say that you want to build some analytics software for a school for some reason, you have the following data in a csv file:

```
name,science,math,history
tony,12,12,12
antony,13,13,13
bantony,14,14,14
```

file name: ch13/student_scores.py

```
input_file = open("data.csv", "r")
score = input_file.readlines()
score = [line.strip() for line in score]

heading = score[0].split(",")
score = score[1:]
total_subjects = len(heading)

marks = {}

for i in range(len(score)):
    sc = score[i].split(",")
    marks[sc[0]] = sum([int(j) for j in sc[1:]])

for name in marks.keys():
    print("%s: %d"%( name, marks[name]))

#TODO print the name of the student who got the maximum aggregare marks
# This part is intentionally kept as homework for the readers.
```

CSV to SQL generator

Let's say that you have a csv file which you want to import into a database and for some reason your db client doesn't support direct import, so you have the task to convert the data in csv file into insert statements.

This is a crude way to convert data, if you are having access to the database client in python, you can write your own importer without having to generate SQL statements.

file name: ch13/csv_to_db.py

```
file_name = "data.csv"
input_file = open(file_name, "r")
output_file = open("data.sql", "w")

csv_lines = input_file.readlines()
csv_lines = [line.strip() for line in csv_lines]

INSRT_STMT = 'INSERT INTO STUDENT(NAME, SCIENCE, MATH, HISTORY) VALUES('

inserts = []

for line in csv_lines[1:]:
    iline = line.split(",")
    insert = INSRT_STMT + '"' + iline[0] + '"'
    for i in iline[1:]:
        insert += ","
        insert += i
    insert+=");\n"
    inserts.append(insert)

output_file.writelines(inserts)
output_file.close()
```

Links

| Previous[73] | Index[74] | —-| —-|

[73] 12-modules-tasks.md
[74] ../SUMMARY.md