Table of Contents

Python Basics

Introduction to Python - Syntax, working with Shell commands, Files, Text Processing, and more...

- Suitable for a one/two day workshop for Python beginners
- Visit Python re(gex)? repo for a book on regular expressions
- Python curated resources for more complete resources list, including tutorials for complete beginners to programming
- For more related resources, visit scripting course and my programming blog https://learnbyexample.github.io

Chapters

- Introduction
  - Installation, Hello World example, Python Interpreter, Python Standard Library
- Number and String data types
  - Numbers, String, Constants, Built-in Operators
- Functions
  - def, print function, range function, type function, Variable Scope
- Getting User input
  - Integer input, Floating point input, String input
- Executing external commands
  - Calling Shell commands, Calling Shell commands with expansion, Getting command output and redirections
- Control Structures
  - Condition checking, if, for, while, continue and break
- Lists
  - Assigning List variables, Slicing and Modifying Lists, Copying Lists, List Methods and Miscellaneous, Looping, List Comprehension, Getting List as user input, Getting random items from list
- Sequence, Set and Dict data types
  - Strings, Tuples, Set, Dictionary
- Text Processing
  - String methods, Regular Expressions, Pattern matching and extraction, Search and Replace, Compiling Regular Expressions, Further Reading on Regular Expressions
- File handling
  - open function, Reading files, Writing to files, Inplace editing with fileinput
- Command line arguments
  - Known number of arguments, Varying number of arguments, Using program name in code,

Contributing

- Please open an issue on github for typos/bugs/suggestions/etc
  - Even for pull requests, open an issue for discussion before submitting PRs
  - or gitter group chat for discussion as well as for help/mentorship
- Share the repo with friends/colleagues, on social media, etc to help reach other learners
- In case you need to reach me, use gitter private chat
  - or mail me at `echo 'bGVhcm5ieWV4YW1wbGUubmV0QGdtYWlsLmNvbQo=' | base64 --decode`

ebook

- Read as ebook on gitbook
- Download ebook for offline reading - link

Acknowledgements

- automatetheboringstuff for getting me started with Python
- /r/learnpython/ - helpful forum for beginners and experienced programmers alike
- stackoverflow - for getting answers to pertinent questions as well as sharpening skills by understanding and answering questions
- Devs and Hackers - helpful slack group
- Weekly Coders, Hackers & All Tech related thread - for suggestions and critique

License

Introduction

- Installation
- Hello World example
- Python Interpreter
- Python Standard Library

From wikipedia)

> Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale

Guido van Rossum is the author of Python programming language, and continues to oversee the Python development process

Installation

- Get Python for your OS from official website - https://www.python.org/
  - Most Linux distributions come with Python by default
- See also this guide for more detail as well as how to set up virtual environment, how to use **pip** (NEVER use **sudo pip** unless you know what you are doing)

- Examples presented here is for **Unix-like systems**, Python version 3 and uses **bash** shell
- You can also run Python code online
  - pythontutor - code execution in Python 2 and 3 versions, visualizing code flow and sample programs are among its features
  - jupyter - web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text
  - ideone - online compiler and debugging tool which allows you to compile source code and execute it online in more than 60 programming languages
  - Python Interpreter shell
- It is assumed that you are familiar with command line. If not, check out this basic tutorial on ryanstutorials and this list of curated resources for Linux

Hello World example

Let's start with simple a Python program and how to run it

```
#!/usr/bin/python3

print("Hello World")
```

The first line has two parts

- `/usr/bin/python3` is the path of Python interpreter
- `#!` called as **shebang)**, directs the program loader to use the interpreter path provided

The third line prints the message `Hello World` with a newline character added by default by the `print` function

Running Python program

You can write the program using text editor like **gedit**, **vim** or other editors
After saving the file, give execute permission and run the program from a terminal

```
$ chmod +x hello_world.py

$ ./hello_world.py
Hello World
```

To find out path and version of Python in your system

```
$ type python3
python3 is /usr/bin/python3

$ python3 --version
Python 3.4.3
```

If you happen to follow a book/tutorial on Python version 2 or coming with Perl experience, it is a common mistake to forget () with `print` function

```
#!/usr/bin/python3

print "Have a nice day"
```

- Depending on type of error, it may be easy to spot the mistake based on error messages printed on executing the program
- In this example, we get the appropriate `Missing parentheses` message

```
$ ./syntax_error.py
  File "./syntax_error.py", line 3
    print "Have a nice day"
                          ^
SyntaxError: Missing parentheses in call to 'print'
```

- single line comments start with `#`
  - `#!` has special meaning only on first line of program
- we will see multiline comments in later chapters

```
#!/usr/bin/python3

# Greeting message
print("Hello World")
```

Further Reading

- Python docs - version 3
- Different ways of executing Python programs
- Where is Python used?
- Python docs - Errors and Exceptions
- Common syntax errors

Python Interpreter

- It is generally used to execute snippets of Python language as a means to learning Python or for debugging purposes
- The prompt is usually `>>>`
- Some of the topics in coming chapters will be complemented with examples using the Python Interpreter
- A special variable `_` holds the result of last printed expression
- One can type part of command and repeatedly press Up arrow key to match commands from history
- Press `Ctrl+l` to clear the screen, keeping any typed command intact
- `exit()` to exit

```
$ python3
Python 3.4.3 (default, Oct 14 2015, 20:28:29)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hi")
hi
>>> abc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'abc' is not defined
>>> num = 5
>>> num
5
>>> 3 + 4
7
>>> 12 + _
19
>>> exit()
```

Further Reading

- Python docs - Using the Python Interpreter
- Python docs - Interpreter example

Python Standard Library

- Python docs - library
- pypi - repository of software for the Python programming language

> The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming.
>
> Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs

Number and String data types

Variable data type is automatically determined by Python. They only need to be assigned some value before using it elsewhere - like print function or part of expression

Numbers

- Integer examples

```
>>> num1 = 7
>>> num2 = 42
>>> total = num1 + num2
>>> print(total)
49
>>> total
49

# no limit to integer precision, only limited by available memory
>>> 34 ** 32
1017010285931541177457962846134113802302590130585 6

# using single / gives floating point output
>>> 9 / 5
1.8

# using double / gives only the integer portion, no rounding
>>> 9 // 5
1

>>> 9 % 5
4
```

- Floating point examples

```
>>> appx_pi = 22 / 7
>>> appx_pi
3.142857142857143

>>> area = 42.16
>>> appx_pi + area
45.30285714285714

>>> num1
7
>>> num1 + area
49.16
```

- the E scientific notation can be used as well

```
>>> sci_num1 = 3.982e5
>>> sci_num2 = 9.32e-1
>>> sci_num1 + sci_num2
398200.932

>>> 2.13e21 + 5.23e22
5.443e+22
```

- Binary numbers are prefixed with `0b` or `0B` (i.e digit 0 followed by lower/upper case letter b)
- Octal numbers are prefixed with `0o` or `0O` (i.e digit 0 followed by lower/upper case letter o)
- Similarly, Hexadecimal numbers are prefixed with `0x` or `0X`

```
>>> bin_num = 0b101
>>> oct_num = 0o12
>>> hex_num = 0xF

>>> bin_num
5
>>> oct_num
10
>>> hex_num
15

>>> oct_num + hex_num
25
```

- `_` can be used between digits for readability
  - introduced in Python v3.6

```
>>> 1_000_000
1000000
>>> 1_00.3_352
100.3352
>>> 0xff_ab1
1047217

# f-strings formatting explained in a later chapter
>>> num = 34 ** 32
>>> print(f'{num:_}')
10_170_102_859_315_411_774_579_628_461_341_138_023_025_901_305_856
```

Further Reading

- Python docs - numbers
- decimal
- fractions
- complex
- Python docs - keywords - do not use these as variables

String

- strings can be declared using single or double quotes
- Use `\` to escape quotes which are part of string itself if the string contains both single and double quotes

```
>>> str1 = 'This is a string'
>>> str1
'This is a string'
>>> greeting = "Hello World!"
>>> greeting
'Hello World!'

>>> weather = "It's a nice and warm day"
>>> weather
"It's a nice and warm day"
>>> print(weather)
It's a nice and warm day

>>> weather = 'It\'s a nice and warm day'
>>> print(weather)
It's a nice and warm day
```

- Escape sequences like newline character `\n` can be used within string declaration

```
>>> colors = 'Blue\nRed\nGreen'
>>> colors
'Blue\nRed\nGreen'

>>> print(colors)
Blue
Red
Green
```

- Use `r` prefix (stands for **raw**) if you do not want escape sequences to be interpreted
- It is commonly used with regular expressions, see Pattern matching and extraction for examples

```
>>> raw_str = r'Blue\nRed\nGreen'
>>> print(raw_str)
Blue\nRed\nGreen

# to see how the string is stored internally
>>> raw_str
'Blue\\nRed\\nGreen'
```

- String concatenation and repetition

```
>>> str1 = 'Hello'
>>> str2 = ' World'
>>> print(str1 + str2)
Hello World

>>> style_char = '-'
>>> style_char * 10
'----------'

>>> word = 'buffalo '
>>> print(word * 8)
buffalo buffalo buffalo buffalo buffalo buffalo buffalo buffalo

# Python v3.6 allows variable interpolation with f-strings
>>> msg = f'{str1} there'
>>> msg
'Hello there'
```

- Triple quoted strings
- like single line strings, `"""` or `'''` can be used as required as well as escape characters using `\`

```
#!/usr/bin/python3

"""
This line is part of multiline comment

This program shows examples of triple quoted strings
"""

# assigning multiple line string to variable
poem = """\
The woods are lovely, dark and deep,
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.
"""

print(poem, end='')
```

- Triple quoted strings also help in documentation, see Docstrings chapter for examples

```
$ ./triple_quoted_string.py
The woods are lovely, dark and deep,
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.
$
```

Further Reading

- Python docs - strings
- Python docs - f-strings - for more examples and caveats
- Python docs - List of Escape Sequences and more info on strings
- Python docs - Binary Sequence Types
- formatting triple quoted strings

Constants

Paraphrased from Python docs - constants

- `None` The sole value of the type `NoneType`
  - `None` is frequently used to represent the absence of a value
- `False` The false value of the `bool` type
- `True` The true value of the `bool` type

- Python docs - Truth Value Testing

```
>>> bool(2)
True
>>> bool(0)
False
>>> bool('')
False
>>> bool('a')
True
```

Built-in Operators

- arithmetic operators
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division (float output)
  - `//` division (integer output, result is not rounded)
  - `**` exponentiation
  - `%` modulo
- string operators
  - `+` string concatenation
  - `*` string repetition
- comparison operators
  - `==` equal to
  - `>` greater than
  - `<` less than
  - `!=` not equal to
  - `>=` greater than or equal to
  - `<=` less than or equal to
- boolean logic
  - `and` logical and
  - `or` logical or
  - `not` logical not
- bitwise operators
  - `&` and
  - `|` or
  - `^` exclusive or
  - `~` invert bits

    - `>>` right shift
    - `<<` left shift
- and many more...

Further Reading

- Python docs - Numeric types - complete list of operations and precedence
- Python docs - String methods

Functions

def

```
#!/usr/bin/python3

# ----- function without arguments -----
def greeting():
    print("-----------------------------")
    print("         Hello World         ")
    print("-----------------------------")

greeting()

# ----- function with arguments -----
def sum_two_numbers(num1, num2):
    total = num1 + num2
    print("{} + {} = {}".format(num1, num2, total))

sum_two_numbers(3, 4)

# ----- function with return value -----
def num_square(num):
    return num * num

my_num = 3
print(num_square(2))
print(num_square(my_num))
```

- The `def` keyword is used to define functions
- Functions have to be defined before use
- A common syntax error is leaving out `:` at end of `def` statement
- Block of code for functions, control structures, etc are distinguished by indented code
  - 4-space indentation is recommended
  - Python docs - Coding Style

- The default `return` value is `None`
- How variables are passed to functions in Python
- `format` is covered in next topic

```
$ ./functions.py
----------------------------
         Hello World
----------------------------
3 + 4 = 7
4
9
```

Default valued arguments

```
#!/usr/bin/python3

# ----- function with default valued argument -----
def greeting(style_char='-'):
    print(style_char * 29)
    print("         Hello World         ")
    print(style_char * 29)

print("Default style")
greeting()

print("\nStyle character *")
greeting('*')

print("\nStyle character =")
greeting(style_char='=')
```

- Often, functions can have a default behavior and if needed changed by passing relevant argument

```
$ ./functions_default_arg_value.py
Default style
----------------------------
         Hello World
----------------------------

Style character *
**************************
         Hello World
**************************

Style character =
==========================
         Hello World
==========================
```

- Triple quoted comment describing function purpose is a usually followed guideline
- To avoid distraction from example code, docstrings for programs and functions won't be generally used in this tutorial
  - See Docstrings chapter for examples and discussion

```
def num_square(num):
    """
    returns square of number
    """

    return num * num
```

Further Reading

There are many more ways to call a function and other types of declarations, refer the below links for more info

- Python docs - defining functions
- Python docs - Built-in Functions

print function

- By default, `print` function adds newline character
- This can be changed by passing our own string to the `end` argument

```
>>> print("hi")
hi
>>> print("hi", end='')
hi>>>
>>> print("hi", end=' !!\n')
hi !!
>>>
```

- The help function can be used to get quick help from interpreter itself
- Press `q` to return back from help page

```
>>> help(print)

Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

- Multiple arguments to `print` function can be passed by `,` separation
- The default `sep` is single space character

```
>>> a = 5
>>> b = 2

>>> print(a+b, a-b)
7 3

>>> print(a+b, a-b, sep=' : ')
7 : 3

>>> print(a+b, a-b, sep='\n')
7
3
```

- When printing variables, the **str** method is called which gives the string representation
- So, explicit conversion is not needed unless concatenation is required

```
>>> greeting = 'Hello World'
>>> print(greeting)
Hello World
>>> num = 42
>>> print(num)
42

>>> print(greeting + '. We are learning Python')
Hello World. We are learning Python
>>> print(greeting, '. We are learning Python', sep='')
Hello World. We are learning Python

>>> print("She bought " + num + " apples")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly

>>> print("She bought " + str(num) + " apples")
She bought 42 apples
```

- As an alternative, use multiple arguments and change `sep` accordingly

```
>>> print("She bought", num, "apples")
She bought 42 apples

>>> items = 15
>>> print("No. of items:", items)
No. of items: 15

>>> print("No. of items:", items, sep='')
No. of items:15
```

- To redirect print output to stderr instead of default stdout, change the `file` argument
- See also sys.exit()

```
>>> import sys
>>> print("Error!! Not a valid input", file=sys.stderr)
Error!! Not a valid input
```

- `str.format()` can be used to style strings and handle multiple variables more elegantly than string concatenation

```
>>> num1 = 42
>>> num2 = 7

>>> '{} + {} = {}'.format(num1, num2, num1 + num2)
'42 + 7 = 49'

# or save formatting in a variable and use wherever needed
>>> op_fmt = '{} + {} = {}'
>>> op_fmt.format(num1, num2, num1 + num2)
'42 + 7 = 49'
>>> op_fmt.format(num1, 29, num1 + 29)
'42 + 29 = 71'

# and of course the expression can be used inside print directly
>>> print('{} + {} = {}'.format(num1, num2, num1 + num2))
42 + 7 = 49
```

- using numbered arguments

```
>>> num1
42
>>> num2
7
>>> print("{0} + {1} * {0} = {2}".format(num1, num2, num1 + num2 * num1))
42 + 7 * 42 = 336
```

- number formatting - specified using optional argument number, followed by `:` and then the formatting style

```
>>> appx_pi = 22 / 7
>>> appx_pi
3.142857142857143

# restricting number of digits after decimal point
# value is rounded off
>>> print("{0:.2f}".format(appx_pi))
3.14
>>> print("{0:.3f}".format(appx_pi))
3.143

# aligning
>>> print("{0:<10.3f} and 5.12".format(appx_pi))
3.143      and 5.12
>>> print("{0:>10.3f} and 5.12".format(appx_pi))
     3.143 and 5.12

# zero filling
>>> print("{0:08.3f}".format(appx_pi))
0003.143
```

- different base

```
>>> print("42 in binary = {:b}".format(42))
42 in binary = 101010
>>> print("42 in octal = {:o}".format(42))
42 in octal = 52
>>> print("241 in hex = {:x}".format(241))
241 in hex = f1

# add # for 0b/0o/0x prefix
>>> print("42 in binary = {:#b}".format(42))
42 in binary = 0b101010

>>> hex_str = "{:x}".format(42)
>>> hex_str
'2a'

# can also use format built-in function
>>> format(42, 'x')
'2a'
>>> format(42, '#x')
'0x2a'

# converting string to int
>>> int(hex_str, base=16)
42
>>> int('0x2a', base=16)
42
```

- similar to the `r` raw string prefix, using `f` prefix allows to represent format strings
  - introduced in Python v3.6
- similar to `str.format()`, the variables/expressions are specified within `{}`

```
>>> num1 = 42
>>> num2 = 7
>>> f'{num1} + {num2} = {num1 + num2}'
'42 + 7 = 49'
>>> print(f'{num1} + {num2} = {num1 + num2}')
42 + 7 = 49

>>> appx_pi = 22 / 7
>>> f'{appx_pi:08.3f}'
'0003.143'

>>> f'{20:x}'
'14'
>>> f'{20:#x}'
'0x14'
```

Further Reading

- Python docs - formatstrings - for more info and examples
- Python docs - f-strings - for more examples and caveats

range function

- By default `start=0` and `step=1`, so they can be skipped or defined as appropriate
  - `range(stop)`
  - `range(start, stop)`
  - `range(start, stop, step)`
- Note that `range` output doesn't include `stop` value - it is always upto `stop` value but not including it
- See Lists chapters for discussion and examples on lists
- Python docs - Ranges - for more info and examples

```
>>> range(5)
range(0, 5)

>>> list(range(5))
[0, 1, 2, 3, 4]

>>> list(range(-2, 2))
[-2, -1, 0, 1]

>>> list(range(1, 15, 2))
[1, 3, 5, 7, 9, 11, 13]

>>> list(range(10, -5, -2))
[10, 8, 6, 4, 2, 0, -2, -4]
```

type function

Useful to check data type of a variable or value

```
>>> type(5)
<class 'int'>

>>> type('Hi there!')
<class 'str'>

>>> type(range(7))
<class 'range'>

>>> type(None)
<class 'NoneType'>

>>> type(True)
<class 'bool'>

>>> arr = list(range(4))
>>> arr
[0, 1, 2, 3]
>>> type(arr)
<class 'list'>
```

Variable Scope

```
#!/usr/bin/python3

def print_num():
    print("Yeehaw! num is visible in this scope, its value is: " + str(num))

num = 25
print_num()
```

- Variables defined before function call are visible within the function scope too
- Python docs - Default Argument Values - see description for when default values are evaluated

```
$ ./variable_scope_1.py
Yeehaw! num is visible in this scope, its value is: 25
```

What happens when a variable declared within a block is used outside of it?

```
#!/usr/bin/python3

def square_of_num(num):
    sqr_num = num * num

square_of_num(5)
print("5 * 5 = {}".format(sqr_num))
```

- Here, `sqr_num` is declared inside `square_of_num` function and not accessible outside the block

```
$ ./variable_scope_2.py
Traceback (most recent call last):
  File "./variable_scope_2.py", line 7, in <module>
    print("5 * 5 = {}".format(sqr_num))
NameError: name 'sqr_num' is not defined
```

One way to overcome this is to use the `global` keyword

```
#!/usr/bin/python3

def square_of_num(num):
    global sqr_num
    sqr_num = num * num

square_of_num(5)
print("5 * 5 = {}".format(sqr_num))
```

- Now, we can access `sqr_num` even outside the function definition

```
$ ./variable_scope_3.py
5 * 5 = 25
```

If a variable name is same outside and within function definition, the one inside the function will stay local to the block and not affect the one outside of it

```
#!/usr/bin/python3

sqr_num = 4

def square_of_num(num):
    sqr_num = num * num
    print("5 * 5 = {}".format(sqr_num))

square_of_num(5)
print("Whoops! sqr_num is still {}!".format(sqr_num))
```

- Note that using `global sqr_num` will affect the `sqr_num` variable outside the function

```
$ ./variable_scope_4.py
5 * 5 = 25
Whoops! sqr_num is still 4!
```

Further Reading

- Python docs - scope example
- Python docs - global statement

Getting User input

Integer input

```
#!/usr/bin/python3

usr_ip = input("Enter an integer number: ")

# Need to explicitly convert input string to desired type
usr_num = int(usr_ip)
sqr_num = usr_num * usr_num

print("Square of entered number is: {}".format(sqr_num))
```

- Let us test the program by giving an integer number and a string
- Python docs - integer-literals

```
$ ./user_input_int.py
Enter an integer number: 23
Square of entered number is: 529

$ ./user_input_int.py
Enter an integer number: abc
Traceback (most recent call last):
  File "./user_input_int.py", line 6, in <module>
    usr_num = int(usr_ip)
ValueError: invalid literal for int() with base 10: 'abc'
```

Floating point input

```
#!/usr/bin/python3

usr_ip = input("Enter a floating point number: ")

# Need to explicitly convert input string to desired type
usr_num = float(usr_ip)
sqr_num = usr_num * usr_num

# Limit the number of digits after decimal points to 2
print("Square of entered number is: {0:.2f}".format(sqr_num))
```

- The E scientific notation can also be used when required
- Python docs - floating-point-literals
- Python docs - floatingpoint

```
$ ./user_input_float.py
Enter a floating point number: 3.232
Square of entered number is: 10.45

$ ./user_input_float.py
Enter a floating point number: 42.7e5
Square of entered number is: 18232900000000.00

$ ./user_input_float.py
Enter a floating point number: abc
Traceback (most recent call last):
  File "./user_input_float.py", line 6, in <module>
    usr_num = float(usr_ip)
ValueError: could not convert string to float: 'abc'
```

String input

```
#!/usr/bin/python3

usr_name  = input("Hi there! What's your name? ")
usr_color = input("And your favorite color is? ")

print("{}, I like the {} color too".format(usr_name, usr_color))
```

- No need any type conversion for string and no newline character to be taken care unlike Perl

```
$ ./user_input_str.py
Hi there! What's your name? learnbyexample
And your favorite color is? blue
learnbyexample, I like the blue color too
```

Executing external commands

The sample output shown in this chapter may be different based on your username, working directories, etc

Calling Shell commands

```
#!/usr/bin/python3

import subprocess

# Executing external command 'date'
subprocess.call('date')

# Passing options and arguments to command
print("\nToday is ", end="", flush=True)
subprocess.call(['date', '-u', '+%A'])

# another example
print("\nSearching for 'hello world'", flush=True)
subprocess.call(['grep', '-i', 'hello world', 'hello_world.py'])
```

- `import` statement here is used to load the `subprocess` module, which is part of the Python standard library
- the `call` function from `subprocess` module is one of the ways to execute external commands
- By passing `True` to `flush` argument (default is `False`) we ensure that our message is printed before `subprocess.call`
- For passing arguments, list of strings is passed instead of single string

```
$ ./calling_shell_commands.py
Tue Jun 21 18:35:33 IST 2016

Today is Tuesday

Searching for 'hello world'
print("Hello World")
```

Further Reading

- Python docs - subprocess
- Python docs - os.system
  - difference between os.system and subprocess.call
- Python docs - import statement

Calling Shell commands with expansion

```
#!/usr/bin/python3

import subprocess

# Executing command without shell expansion
print("No shell expansion when shell=False", flush=True)
subprocess.call(['echo', 'Hello $USER'])

# Executing command with shell expansion
print("\nshell expansion when shell=True", flush=True)
subprocess.call('echo Hello $USER', shell=True)

# escape quotes if it is part of command
print("\nSearching for 'hello world'", flush=True)
subprocess.call('grep -i \'hello world\' hello_world.py', shell=True)
```

- By default, `subprocess.call` will not expand shell wildcards, perform command substitution, etc
- This can be overridden by passing `True` value for `shell` argument
- Note that the entire command is now passed as string and not as a list of strings
- Quotes need to be escaped if they clash between command string and quotes within the command itself
- Use `shell=True` only if you are sure of the command being executed, else it could be a security issue
  - Python docs - subprocess.Popen

```
$ ./shell_expansion.py
No shell expansion when shell=False
Hello $USER

shell expansion when shell=True
Hello learnbyexample

Searching for 'hello world'
print("Hello World")
```

- In certain cases, escaping quotes can be avoided by using combination of single/double quotes as shown below

```
# use alternate quotes, like this
subprocess.call('grep -i "hello world" hello_world.py', shell=True)

# or this
subprocess.call("grep -i 'hello world' hello_world.py", shell=True)
```

- Shell command redirections can be used as usual

```
# use variables for clarity and to avoid long strings within call function
cmd = "grep -h 'test' report.log test_list.txt > grep_test.txt"
subprocess.call(cmd, shell=True)
```

Workaround to avoid using shell=True

```
>>> import subprocess, os
>>> subprocess.call(['echo', 'Hello', os.environ.get("USER")])
Hello learnbyexample
0
```

- `os.environ.get("USER")` gives back the value of environment variable `USER`
- `0` is the exit status, meaning success. It is a caveat of python interpreter which displays return value too

Getting command output and redirections

```
#!/usr/bin/python3

import subprocess

# Output includes any error messages also
print("Getting output of 'pwd' command", flush=True)
curr_working_dir = subprocess.getoutput('pwd')
print(curr_working_dir)

# Get status and output of command executed
# Exit status other than '0' is considered as something gone wrong
ls_command = 'ls hello_world.py xyz.py'
print("\nCalling command '{}'".format(ls_command), flush=True)
(ls_status, ls_output) = subprocess.getstatusoutput(ls_command)
print("status: {}\noutput: '{}'".format(ls_status, ls_output))

# Suppress error messages if preferred
# subprocess.call() returns status of command which can be used instead
print("\nCalling command with error msg suppressed", flush=True)
ls_status = subprocess.call(ls_command, shell=True, stderr=subprocess.DEVNULL)
print("status: {}".format(ls_status))
```

- Output of `getstatusoutput()` is of `tuple` data type, more info and examples in later chapters
- `getstatusoutput()` and `getoutput()` are legacy functions
- Use newer functions for more features and secure options
  - Python docs - subprocess.check_output
  - Python docs - subprocess.run

```
$ ./shell_command_output_redirections.py
Getting output of 'pwd' command
/home/learnbyexample/Python/python_programs

Calling command 'ls hello_world.py xyz.py'
status: 2
output: 'ls: cannot access xyz.py: No such file or directory
hello_world.py'

Calling command with error msg suppressed
hello_world.py
status: 2
```

Control Structures

Condition checking

- simple and combination of tests

```
>>> num = 5
>>> num > 2
True
>>> num > 3 and num <= 5
True
>>> 3 < num <= 5
True
>>> num % 3 == 0 or num % 5 == 0
True

>>> fav_fiction = 'Harry Potter'
>>> fav_detective = 'Sherlock Holmes'
>>> fav_fiction == fav_detective
False
>>> fav_fiction == "Harry Potter"
True
```

- Testing variable or value by themselves

```
>>> bool(num)
True
>>> bool(fav_detective)
True
>>> bool(3)
True
>>> bool(0)
False
>>> bool("")
False
>>> bool(None)
False

>>> if -1:
...     print("-1 evaluates to True in condition checking")
...
-1 evaluates to True in condition checking
```

- The use of `in` operator in condition checking

Compare this way of checking

```
>>> def num_chk(n):
...     if n == 10 or n == 21 or n == 33:
...         print("Number passes condition")
...     else:
...         print("Number fails condition")
...
>>> num_chk(10)
Number passes condition
>>> num_chk(12)
Number fails condition
```

vs this one

```
>>> def num_chk(n):
...     if n in (10, 21, 33):
...         print("Number passes condition")
...     else:
...         print("Number fails condition")
...
>>> num_chk(12)
Number fails condition
>>> num_chk(10)
Number passes condition
```

- `(10, 21, 33)` is a tuple data type, will be covered in later chapters
- Python docs - Truth Value Testing

if

```
#!/usr/bin/python3

num = 45

# only if
if num > 25:
    print("Hurray! {} is greater than 25".format(num))

# if-else
if num % 2 == 0:
    print("{} is an even number".format(num))
else:
    print("{} is an odd number".format(num))

# if-elif-else
# any number of elif can be used
if num < 0:
    print("{} is a negative number".format(num))
elif num > 0:
    print("{} is a positive number".format(num))
else:
    print("{} is neither postive nor a negative number".format(num))
```

- Block of code for functions, control structures, etc are distinguished by indented code
  - 4-space indentation is recommended
  - Python docs - Coding Style
- A common syntax error is leaving out `:` at end of control structure statements
- Using `()` around conditions is optional
- indented block can have any number of statements, including blank lines

```
$ ./if_elif_else.py
Hurray! 45 is greater than 25
45 is an odd number
45 is a positive number
```

if-else as conditional operator

```
#!/usr/bin/python3

num = 42

num_type = 'even' if num % 2 == 0 else 'odd'
print("{} is an {} number".format(num, num_type))
```

- Python doesn't have `?:` conditional operator like many other languages
- Using `if-else` in single line like in this example is one workaround
- More ways of simulating ternary conditional operator

```
$ ./if_else_oneliner.py
42 is an even number
```

for

```
#!/usr/bin/python3

number = 9
for i in range(1, 5):
    mul_table = number * i
    print("{} * {} = {}".format(number, i, mul_table))
```

- traditional iteration based loop can be written using `range` function
  - recall that by default `start=0`, `step=1` and `stop` value is not inclusive
- iterating over variables like list, tuples, etc will be covered in later chapters
- Python docs - itertools

```
$ ./for_loop.py
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
```

while

```
#!/usr/bin/python3

# continuously ask user input till it is a positive integer
usr_string = 'not a number'
while not usr_string.isnumeric():
    usr_string = input("Enter a positive integer: ")
```

- while loop allows us to execute block of statements until a condition is satisfied
- Python docs - string methods

```
$ ./while_loop.py
Enter a positive integer: abc
Enter a positive integer: 1.2
Enter a positive integer: 23
$
```

continue and break

The `continue` and `break` keywords are used to change the normal flow of loops on certain conditions

continue - skip rest of statements in the loop and start next iteration

```
#!/usr/bin/python3

prev_num = 0
curr_num = 0
print("The first ten numbers in fibonacci sequence: ", end='')

for num in range(10):
    print(curr_num, end=' ')

    if num == 0:
        curr_num = 1
        continue

    temp = curr_num
    curr_num = curr_num + prev_num
    prev_num = temp

print("")
```

- `continue` can be placed anywhere in a loop block without having to worry about complicated code flow
- this example is just to show use of `continue`, check this for a more Pythonic way

```
$ ./loop_with_continue.py
The first ten numbers in fibonacci sequence: 0 1 1 2 3 5 8 13 21 34
```

break - skip rest of statements in the loop (if any) and exit loop

```
#!/usr/bin/python3

import random

while True:
    # as with range() function, 500 is not inclusive
    random_int = random.randrange(500)
    if random_int % 4 == 0 and random_int % 6 == 0:
        break
print("Random number divisible by 4 and 6: {}".format(random_int))
```

- `while True:` is generally used as infinite loop
- **randrange** has similar `start, stop, step` arguments as range
- Python docs - random

```
$ ./loop_with_break.py
Random number divisible by 4 and 6: 168
$ ./loop_with_break.py
Random number divisible by 4 and 6: 216
$ ./loop_with_break.py
Random number divisible by 4 and 6: 24
```

The while_loop.py example can be re-written using `break`

```
>>> while True:
        usr_string = input("Enter a positive integer: ")
        if usr_string.isnumeric():
            break

Enter a positive integer: a
Enter a positive integer: 3.14
Enter a positive integer: 1
>>>
```

- in case of nested loops, `continue` and `break` only affect the immediate parent loop

- Python docs - else clauses on loops

Lists

Assigning List variables

- Simple lists and retrieving list elements

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels
['a', 'e', 'i', 'o', 'u']
>>> vowels[0]
'a'
>>> vowels[2]
'i'
>>> vowels[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> even_numbers = list(range(2, 11, 2))
>>> even_numbers
[2, 4, 6, 8, 10]
>>> even_numbers[-1]
10
>>> even_numbers[-2]
8
>>> even_numbers[-15]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- Mix of data types and multi-dimensional lists

```
>>> student = ['learnbyexample', 2016, 'Linux, Vim, Python']
>>> print(student)
['learnbyexample', 2016, 'Linux, Vim, Python']

>>> list_2D = [[1, 3, 2, 10], [1.2, -0.2, 0, 2]]
>>> list_2D[0][0]
1
>>> list_2D[1][0]
1.2
```

- Python docs - lists

Slicing and Modifying Lists

- Like the `range()` function, list index has `start:stop:step` format, `stop` value being non-inclusive
- stackoverflow - explain slice notation

```
>>> books = ['Harry Potter', 'Sherlock Holmes', 'To Kill a Mocking Bird']
>>> books[2] = "Ender's Game"
>>> print(books)
['Harry Potter', 'Sherlock Holmes', "Ender's Game"]

>>> prime = [2, 3, 5, 7, 11]
>>> prime[2:4]
[5, 7]
>>> prime[:3]
[2, 3, 5]
>>> prime[3:]
[7, 11]

>>> prime[-1]
11
>>> prime[-1:]
[11]
>>> prime[-2:]
[7, 11]


>>> prime[::1]
[2, 3, 5, 7, 11]
>>> prime[::2]
[2, 5, 11]
>>> prime[3:1:-1]
[7, 5]
>>> prime[::-1]
[11, 7, 5, 3, 2]
>>> prime[:]
[2, 3, 5, 7, 11]
```

- when `start` and `stop` values are same
- Useful when they are generated programmatically, see text processing exercise for example

```
>>> nums = [1.2, -0.2, 0, 2]
>>> nums[0:0]
[]
>>> nums[2:2]
[]
>>> nums[-1:-1]
[]
>>> nums[21:21]
[]
```

- The indexing format can be used to extract from list variable or modify itself

```
>>> nums = [1.2, -0.2, 0, 2]
>>> nums[:2] = [1]
>>> nums
[1, 0, 2]

>>> nums = [1.2, -0.2, 0, 2, 4, 23]
>>> nums[:5:2] = [1, 4, 3]
>>> nums
[1, -0.2, 4, 2, 3, 23]

>>> nums = [1, 2, 3, 23]
>>> nums[::-1] = [1, 4, 5, 2]
>>> nums
[2, 5, 4, 1]
```

- helps to modify a list without changing `id`, which is useful if the variable name is referenced elsewhere (see next section)

```
>>> id(nums)
140598790579336
>>> nums[:] = [1, 2, 5, 4.3]
>>> nums
[1, 2, 5, 4.3]
>>> id(nums)
140598790579336

# assignment without using [:] will change id
>>> nums = [1.2, -0.2, 0, 2]
>>> id(nums)
140598782943752
```

Copying Lists

- Variables in Python contain reference to objects
- For example, when an integer variable is modified, the variable's reference is updated with new object
- the id() function returns the "identity" of an object
- For variables referring to immutable types like integer and strings, this distinction usually doesn't cause confusion in their usage

```
>>> a = 5
>>> id(a)
10105952
>>> a = 10
>>> id(a)
10106112

>>> b = a
>>> id(b)
10106112
>>> b = 4
>>> b
4
>>> a
10
>>> id(b)
10105920
```

- But for variables referring to mutable types like lists, it is important to know how variables are copied and passed to functions
- When an element of list variable is modified, it does so by changing the value (mutation) of object at that index

```
>>> a = [1, 2, 5, 4.3]
>>> a
[1, 2, 5, 4.3]
>>> b = a
>>> b
[1, 2, 5, 4.3]
>>> id(a)
140684757120264
>>> id(b)
140684757120264
>>> b[0] = 'xyz'
>>> b
['xyz', 2, 5, 4.3]
>>> a
['xyz', 2, 5, 4.3]
```

- avoid copying lists using indexing format, it works for 1D lists but not for higher dimensions

```
>>> prime = [2, 3, 5, 7, 11]
>>> b = prime[:]
>>> id(prime)
140684818101064
>>> id(b)
140684818886024
>>> b[0] = 'a'
>>> b
['a', 3, 5, 7, 11]
>>> prime
[2, 3, 5, 7, 11]

>>> list_2D = [[1, 3, 2, 10], [1.2, -0.2, 0, 2]]
>>> a = list_2D[:]
>>> id(list_2D)
140684818102344
>>> id(a)
140684818103048
>>> a
[[1, 3, 2, 10], [1.2, -0.2, 0, 2]]
>>> a[0][0] = 'a'
>>> a
[['a', 3, 2, 10], [1.2, -0.2, 0, 2]]
>>> list_2D
[['a', 3, 2, 10], [1.2, -0.2, 0, 2]]
```

- use the copy module instead

```
>>> import copy
>>> list_2D = [[1, 3, 2, 10], [1.2, -0.2, 0, 2]]
>>> c = copy.deepcopy(list_2D)
>>> c[0][0] = 'a'
>>> c
[['a', 3, 2, 10], [1.2, -0.2, 0, 2]]
>>> list_2D
[[1, 3, 2, 10], [1.2, -0.2, 0, 2]]
```

List Methods and Miscellaneous

- adding elements to list

```
>>> books = []
>>> books
[]
>>> books.append('Harry Potter')
>>> books
['Harry Potter']

>>> even_numbers
[2, 4, 6, 8, 10]
>>> even_numbers += [12, 14, 16, 18, 20]
>>> even_numbers
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

>>> even_numbers = [2, 4, 6, 8, 10]
>>> even_numbers.extend([12, 14, 16, 18, 20])
>>> even_numbers
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

>>> a = [[1, 3], [2, 4]]
>>> a.extend([[5, 6]])
>>> a
[[1, 3], [2, 4], [5, 6]]
```

- deleting elements from a list - based on index

```
>>> prime = [2, 3, 5, 7, 11]
>>> prime.pop()
11
>>> prime
[2, 3, 5, 7]
>>> prime.pop(0)
2
>>> prime
[3, 5, 7]

>>> list_2D = [[1, 3, 2, 10], [1.2, -0.2, 0, 2]]
>>> list_2D[0].pop(0)
1
>>> list_2D
[[3, 2, 10], [1.2, -0.2, 0, 2]]

>>> list_2D.pop(1)
[1.2, -0.2, 0, 2]
>>> list_2D
[[3, 2, 10]]
```

- using del to delete elements

```
>>> nums = [1.2, -0.2, 0, 2, 4, 23]
>>> del nums[1]
>>> nums
[1.2, 0, 2, 4, 23]
# can use slicing notation as well
>>> del nums[1:4]
>>> nums
[1.2, 23]

>>> list_2D = [[1, 3, 2, 10], [1.2, -0.2, 0, 2]]
>>> del list_2D[0][1]
>>> list_2D
[[1, 2, 10], [1.2, -0.2, 0, 2]]

>>> del list_2D[0]
>>> list_2D
[[1.2, -0.2, 0, 2]]
```

- clearing list

```
>>> prime = [2, 3, 5, 7, 11]
>>> prime.clear()
>>> prime
[]
```

- deleting elements from a list - based on value

```
>>> even_numbers = [2, 4, 6, 8, 10]
>>> even_numbers.remove(8)
>>> even_numbers
[2, 4, 6, 10]
>>> even_numbers.remove(12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

- inserting elements at a particular index

```
>>> books = ['Harry Potter', 'Sherlock Holmes', 'To Kill a Mocking Bird']
>>> books.insert(2, "The Martian")
>>> books
['Harry Potter', 'Sherlock Holmes', 'The Martian', 'To Kill a Mocking Bird']
```

- get index of an element

```
>>> even_numbers = [2, 4, 6, 8, 10]
>>> even_numbers.index(6)
2
>>> even_numbers.index(12)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 12 is not in list
```

- checking if an element is present

```
>>> prime = [2, 3, 5, 7, 11]
>>> 3 in prime
True
>>> 13 in prime
False
```

- sorting

```
>>> a = [1, 5.3, 321, 0, 1, 2]
>>> a.sort()
>>> a
[0, 1, 1, 2, 5.3, 321]

>>> a = [1, 5.3, 321, 0, 1, 2]
>>> a.sort(reverse=True)
>>> a
[321, 5.3, 2, 1, 1, 0]
```

Sort based on key, for example based on string length
lambda expressions are helpful to pass custom single expression, say sort based on second letter

```
>>> words = ['fuliginous', 'crusado', 'morello', 'irk', 'seam']
>>> words.sort(key=len)
>>> words
['irk', 'seam', 'crusado', 'morello', 'fuliginous']

>>> words.sort(key=lambda x: x[1])
>>> words
['seam', 'morello', 'irk', 'crusado', 'fuliginous']
```

If original list should not be changed, use sorted function

```
>>> nums = [-1, 34, 0.2, -4, 309]
>>> nums_desc = sorted(nums, reverse=True)
>>> nums_desc
[309, 34, 0.2, -1, -4]

>>> sorted(nums, key=abs)
[0.2, -1, -4, 34, 309]
```

- `min` and `max`

```
>>> a = [321, 899.232, 5.3, 2, 1, -1]
>>> min(a)
-1
>>> max(a)
899.232
```

- Number of times an item is present

```
>>> nums = [15, 99, 19, 382, 43, 19]
>>> nums.count(99)
1
>>> nums.count(19)
2
>>> nums.count(23)
0
```

- reverse list in place

```
>>> prime = [2, 3, 5, 7, 11]
>>> id(prime)
140684818102088
>>> prime.reverse()
>>> prime
[11, 7, 5, 3, 2]
>>> id(prime)
140684818102088

>>> a = [1, 5.3, 321, 0, 1, 2]
>>> id(a)
140684818886024
>>> a = a[::-1]
>>> a
[2, 1, 0, 321, 5.3, 1]
>>> id(a)
140684818102664
```

- len function to get size of lists

```
>>> prime
[2, 3, 5, 7, 11]
>>> len(prime)
5

>>> s = len(prime) // 2
>>> prime[:s]
[2, 3]
>>> prime[s:]
[5, 7, 11]
```

- summing numeric lists

```
>>> a
[321, 5.3, 2, 1, 1, 0]
>>> sum(a)
330.3
```

- all and any functions

```
>>> conditions = [True, False, True]
>>> all(conditions)
False
>>> any(conditions)
True

>>> conditions[1] = True
>>> all(conditions)
True

>>> a = [321, 5.3, 2, 1, 1, 0]
>>> all(a)
False
>>> any(a)
True
```

- comparing lists

```
>>> prime
[2, 3, 5, 7, 11]
>>> a = [4, 2]
>>> prime == a
False

>>> prime == [2, 3, 5, 11, 7]
False
>>> prime == [2, 3, 5, 7, 11]
True
```

Further Reading

- Python docs - more on lists
- Python docs - collections

Looping

```
#!/usr/bin/python3

numbers = [2, 12, 3, 25, 624, 21, 5, 9, 12]
odd_numbers  = []
even_numbers = []

for num in numbers:
    odd_numbers.append(num) if(num % 2) else even_numbers.append(num)

print("numbers:      {}".format(numbers))
print("odd_numbers:  {}".format(odd_numbers))
print("even_numbers: {}".format(even_numbers))
```

- usually, it is enough to deal with every element of list without needing index of elements

```
$ ./list_looping.py
numbers:      [2, 12, 3, 25, 624, 21, 5, 9, 12]
odd_numbers:  [3, 25, 21, 5, 9]
even_numbers: [2, 12, 624, 12]
```

- use `enumerate()` if both index and element is needed

```
#!/usr/bin/python3

north_dishes = ['Aloo tikki', 'Baati', 'Khichdi', 'Makki roti', 'Poha']

print("My favorite North Indian dishes:")
for idx, item in enumerate(north_dishes):
    print("{}. {}".format(idx + 1, item))
```

- In this case, we get a tuple every iteration consisting of a count (default value 0) and an item from the list
- Python docs - enumerate

```
$ ./list_looping_enumeration.py
My favorite North Indian dishes:
1. Aloo tikki
2. Baati
3. Khichdi
4. Makki roti
5. Poha
```

- a start value can also be specified

```
>>> north_dishes = ['Aloo tikki', 'Baati', 'Khichdi', 'Makki roti', 'Poha']
>>> for idx, item in enumerate(north_dishes, start=1):
...     print(idx, item, sep='. ')
...
1. Aloo tikki
2. Baati
3. Khichdi
4. Makki roti
5. Poha
```

- use `zip()` to iterate over two or more lists simultaneously
- Python docs - zip

```
>>> odd = [1, 3, 5]
>>> even = [2, 4, 6]
>>> for i, j in zip(odd, even):
...     print(i + j)
...
3
7
11
```

List Comprehension

```
#!/usr/bin/python3

import time

numbers = list(range(1,100001))
fl_square_numbers = []

# reference time
t0 = time.perf_counter()

# ------------ for loop ------------
for num in numbers:
    fl_square_numbers.append(num * num)

# reference time
t1 = time.perf_counter()

# ------- list comprehension -------
lc_square_numbers = [num * num for num in numbers]

# performance results
t2 = time.perf_counter()
fl_time = t1 - t0
lc_time = t2 - t1
improvement = (fl_time - lc_time) / fl_time * 100

print("Time with for loop:           {:.4f}".format(fl_time))
print("Time with list comprehension: {:.4f}".format(lc_time))
print("Improvement:                  {:.2f}%".format(improvement))

if fl_square_numbers == lc_square_numbers:
    print("\nfl_square_numbers and lc_square_numbers are equivalent")
else:
    print("\nfl_square_numbers and lc_square_numbers are NOT equivalent")
```

- List comprehensions is a Pythonic way for some of the common looping constructs
- Usually is a more readable and time saving option than loops
- In this example, not having to call `append()` method also saves lot of time in case of list comprehension
- Time values in this example is indicative and not to be taken as absolute
    - It usually varies even between two runs, let alone different machines

```
$ ./list_comprehension.py
Time with for loop:           0.0142
Time with list comprehension: 0.0062
Improvement:                  56.36%

fl_square_numbers and lc_square_numbers are equivalent
```

- conditional list comprehension

```
# using if-else conditional in list comprehension
numbers = [2, 12, 3, 25, 624, 21, 5, 9, 12]
odd_numbers  = []
even_numbers = []
[odd_numbers.append(num) if(num % 2) else even_numbers.append(num) for num in number
s]

# or a more simpler and readable approach
numbers = [2, 12, 3, 25, 624, 21, 5, 9, 12]
odd_numbers  = [num for num in numbers if num % 2]
even_numbers = [num for num in numbers if not num % 2]
```

- zip example

```
>>> p = [1, 3, 5]
>>> q = [3, 214, 53]
>>> [i+j for i,j in zip(p, q)]
[4, 217, 58]
>>> [i*j for i,j in zip(p, q)]
[3, 642, 265]
```

use generator expressions if sequence needs to be passed onto another function

```
>>> sum(i*j for i,j in zip(p, q))
910
```

Further Reading

For more examples, including nested loops, check these

- Python docs - list comprehensions
- Python List Comprehensions: Explained Visually
- are list comprehensions and functional functions faster than for loops
- Python docs - perf_counter
  - understanding perf_counter and process_time

- Python docs - timeit

Getting List as user input

```
>>> b = input('Enter strings separated by space: ').split()
Enter strings separated by space: foo bar baz
>>> b
['foo', 'bar', 'baz']

>>> nums = [int(n) for n in input('Enter numbers separated by space: ').split()]
Enter numbers separated by space: 1 23 5
>>> nums
[1, 23, 5]

>>> ip_str = input('Enter prime numbers separated by comma: ')
Enter prime numbers separated by comma: 3,5,7
>>> primes = [int(n) for n in ip_str.split(',')]
>>> primes
[3, 5, 7]
```

- Since user input is all treated as string, need to process based on agreed delimiter and required data type

Getting random items from list

- Get a random item

```
>>> import random
>>> a = [4, 5, 2, 76]
>>> random.choice(a)
76
>>> random.choice(a)
4
```

- Randomly re-arrange items of list

```
>>> random.shuffle(a)
>>> a
[5, 2, 76, 4]
```

- Get random slice of list, doesn't modify the list variable

```
>>> random.sample(a, k=3)
[76, 2, 5]

>>> random.sample(range(1000), k=5)
[68, 203, 15, 757, 580]
```

- Get random items from list without repetition by creating an iterable using Python docs - iter function
- The difference from simply using shuffled list is that this avoids the need to maintain a separate index counter and automatic exception raised if it goes out of range

```
>>> nums = [1, 3, 6, -12, 1.2, 3.14]
>>> random.shuffle(nums)
>>> nums_iter = iter(nums)
>>> print(next(nums_iter))
3.14
>>> print(next(nums_iter))
1.2
>>> for n in nums_iter:
...     print(n)
...
1
3
-12
6
>>> print(next(nums_iter))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- Python docs - random for more info
  - new in version 3.6 - random.choices
- Python docs - next
- See also yield

Sequence, Set and Dict data types

We have already seen Sequence types in previous chapters - strings, ranges and lists. Tuple is another sequence type
We'll see some more operations on strings followed by Tuple, Set and Dict in this chapter

Strings

- The indexing we saw for lists can be applied to strings as well
  - As strings are immutable, they can't be modified like lists though

```
>>> book = "Alchemist"
>>> book[0]
'A'
>>> book[3]
'h'
>>> book[-1]
't'
>>> book[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range

>>> book[2:6]
'chem'
>>> book[:5]
'Alche'
>>> book[5:]
'mist'
>>> book[::-1]
'tsimehclA'
>>> book[:]
'Alchemist'

>>> list(book)
['A', 'l', 'c', 'h', 'e', 'm', 'i', 's', 't']

>>> import string
>>> string.ascii_lowercase[:10]
'abcdefghij'
>>> list(string.digits)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

- looping over strings

```
>>> book
'Alchemist'

>>> for char in book:
...     print(char)
...
A
l
c
h
e
m
i
s
t
```

- Other operations

```
>>> book
'Alchemist'

>>> len(book)
9

>>> book.index('A')
0
>>> book.index('t')
8

>>> 'A' in book
True
>>> 'B' in book
False
>>> 'z' not in book
True

>>> min('zealous')
'a'
>>> max('zealous')
'z'
```

Tuples

- Tuples are similar to lists but immutable and useful in other ways too
- Individual elements can be both mutable/immutable

```
>>> north_dishes = ('Aloo tikki', 'Baati', 'Khichdi', 'Makki roti', 'Poha')
>>> north_dishes
('Aloo tikki', 'Baati', 'Khichdi', 'Makki roti', 'Poha')

>>> north_dishes[0]
'Aloo tikki'
>>> north_dishes[-1]
'Poha'
>>> north_dishes[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range

>>> north_dishes[::-1]
('Poha', 'Makki roti', 'Khichdi', 'Baati', 'Aloo tikki')

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

- Example operations

```
>>> 'roti' in north_dishes
False
>>> 'Makki roti' in north_dishes
True

>>> len(north_dishes)
5

>>> min(north_dishes)
'Aloo tikki'
>>> max(north_dishes)
'Poha'

>>> for dish in north_dishes:
...     print(dish)
...
Aloo tikki
Baati
Khichdi
Makki roti
Poha
```

- Tuple is handy for multiple variable assignment and returning more than one value in functions
  - We have already seen example when using `enumerate` for iterating over lists

```
>>> a = 5
>>> b = 20
>>> a, b = b, a
>>> a
20
>>> b
5

>>> c = 'foo'
>>> a, b, c = c, a, b
>>> a
'foo'
>>> b
20
>>> c
5

>>> def min_max(arr):
...     return min(arr), max(arr)
...
>>> min_max([23, 53, 1, -34, 9])
(-34, 53)
```

- using `()` is not always required

```
>>> words = 'day', 'night'
>>> words
('day', 'night')

>>> coordinates = ((1,2), (4,3), (92,3))
>>> coordinates
((1, 2), (4, 3), (92, 3))

>>> prime = [2, 3, 5, 7, 11]
>>> prime_tuple = tuple((idx + 1, num) for idx, num in enumerate(prime))
>>> prime_tuple
((1, 2), (2, 3), (3, 5), (4, 7), (5, 11))
```

- converting other types to tuples
  - similar to `list()`

```
>>> tuple('books')
('b', 'o', 'o', 'k', 's')

>>> a = [321, 899.232, 5.3, 2, 1, -1]
>>> tuple(a)
(321, 899.232, 5.3, 2, 1, -1)
```

- data types can be mixed and matched in different ways

```
>>> a = [(1,2), ['a', 'b'], ('good', 'bad')]
>>> a
[(1, 2), ['a', 'b'], ('good', 'bad')]

>>> b = ((1,2), ['a', 'b'], ('good', 'bad'))
>>> b
((1, 2), ['a', 'b'], ('good', 'bad'))
```

- Python docs - tuple
- Python docs - tuple tutorial

Set

- Set is unordered collection of objects
- Mutable data type
- Typically used to maintain unique sequence, perform Set operations like intersection, union, difference, symmetric difference, etc

```
>>> nums = {3, 2, 5, 7, 1, 6.3}
>>> nums
{1, 2, 3, 5, 6.3, 7}

>>> primes = {3, 2, 11, 3, 5, 13, 2}
>>> primes
{2, 3, 11, 13, 5}

>>> nums.union(primes)
{1, 2, 3, 5, 6.3, 7, 11, 13}

>>> primes.difference(nums)
{11, 13}
>>> nums.difference(primes)
{1, 6.3, 7}
```

- Example operations

```
>>> len(nums)
6

>>> nums[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing

>>> book
'Alchemist'
>>> set(book)
{'i', 'l', 's', 'A', 'e', 'h', 'm', 't', 'c'}
>>> set([1, 5, 3, 1, 9])
{1, 9, 3, 5}
>>> list(set([1, 5, 3, 1, 9]))
[1, 9, 3, 5]

>>> nums = {1, 2, 3, 5, 6.3, 7}
>>> nums
{1, 2, 3, 5, 6.3, 7}
>>> nums.pop()
1
>>> nums
{2, 3, 5, 6.3, 7}

>>> nums.add(1)
>>> nums
{1, 2, 3, 5, 6.3, 7}

>>> 6.3 in nums
True

>>> for n in nums:
...     print(n)
...
1
2
3
5
6.3
7
```

- Python docs - set
- Python docs - frozenset

- set tutorial

Dictionary

- `dict` types can be thought of as unordered list of `key:value` pairs or a named list of items
- up to Python v3.5 (and some implementations of v3.6) do not retain order of insertion of dict elements

```
>>> marks = {'Rahul' : 86, 'Ravi' : 92, 'Rohit' : 75}
>>> marks
{'Ravi': 92, 'Rohit': 75, 'Rahul': 86}

>>> fav_books = {}
>>> fav_books['fantasy']   = 'Harry Potter'
>>> fav_books['detective'] = 'Sherlock Holmes'
>>> fav_books['thriller']  = 'The Da Vinci Code'
>>> fav_books
{'thriller': 'The Da Vinci Code', 'fantasy': 'Harry Potter', 'detective': 'Sherlock
Holmes'}

>>> marks.keys()
dict_keys(['Ravi', 'Rohit', 'Rahul'])

>>> fav_books.values()
dict_values(['The Da Vinci Code', 'Harry Potter', 'Sherlock Holmes'])
```

- looping and printing

```
>>> for book in fav_books.values():
...     print(book)
...
The Da Vinci Code
Harry Potter
Sherlock Holmes

>>> for name, mark in marks.items():
...     print(name, mark, sep=': ')
...
Ravi: 92
Rohit: 75
Rahul: 86

>>> import pprint
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(fav_books)
{   'detective': 'Sherlock Holmes',
    'fantasy': 'Harry Potter',
    'thriller': 'The Da Vinci Code'}
```

- modifying dicts and example operations

```
>>> marks
{'Ravi': 92, 'Rohit': 75, 'Rahul': 86}
>>> marks['Rajan'] = 79
>>> marks
{'Ravi': 92, 'Rohit': 75, 'Rahul': 86, 'Rajan': 79}

>>> del marks['Ravi']
>>> marks
{'Rohit': 75, 'Rahul': 86, 'Rajan': 79}

>>> len(marks)
3

>>> fav_books
{'thriller': 'The Da Vinci Code', 'fantasy': 'Harry Potter', 'detective': 'Sherlock
Holmes'}
>>> "fantasy" in fav_books
True
>>> "satire" in fav_books
False
```

- dict made up of lists and using random module
- any change in the individual lists will also reflect in dict

- output of `keys()` method has to be changed to Sequence types like `list` or `tuple` to pass on to `random.choice`

```
>>> north = ['aloo tikki', 'baati', 'khichdi', 'makki roti', 'poha']
>>> south = ['appam', 'bisibele bath', 'dosa', 'koottu', 'sevai']
>>> west = ['dhokla', 'khakhra', 'modak', 'shiro', 'vada pav']
>>> east = ['hando guri', 'litti', 'momo', 'rosgulla', 'shondesh']
>>> dishes = {'North': north, 'South': south, 'West': west, 'East': east}

>>> rand_zone = random.choice(tuple(dishes.keys()))
>>> rand_dish = random.choice(dishes[rand_zone])
>>> print("Try the '{}' speciality '{}' today".format(rand_zone, rand_dish))
Try the 'East' speciality 'rosgulla' today
```

- From Python v3.7 onwards, dict implementation will retain insertion order
  - some implementations like the reference CPython implementation for v3.6 also retains the insertion order

```
>>> marks = {'Rahul' : 86, 'Ravi' : 92, 'Rohit' : 75, 'Rajan': 79}
>>> marks
{'Rahul': 86, 'Ravi': 92, 'Rohit': 75, 'Rajan': 79}

>>> for name, mark in marks.items():
...     print(f'{name:5s}: {mark}')
...
Rahul: 86
Ravi : 92
Rohit: 75
Rajan: 79

>>> del marks['Ravi']
>>> marks
{'Rahul': 86, 'Rohit': 75, 'Rajan': 79}

>>> marks['Ranjit'] = 65
>>> marks
{'Rahul': 86, 'Rohit': 75, 'Rajan': 79, 'Ranjit': 65}
```

Further Reading

- Python docs - dict
- Python docs - pprint
- detailed tutorial on dict
- Using dict to eliminate duplicates while retaining order

Text Processing

String methods

- translate string characters
  - `str.maketrans()` to get translation table
  - `translate()` to perform the string mapping based on translation table
- the first argument to `maketrans()` is string characters to be replaced, the second is characters to replace with and the third is characters to be mapped to `None`
- character translation examples

```
>>> greeting = '===== Have a great day ====='
>>> greeting.translate(str.maketrans('=', '-'))
'----- Have a great day -----'

>>> greeting = '===== Have a great day!! ====='
>>> greeting.translate(str.maketrans('=', '-', '!'))
'----- Have a great day -----'

>>> import string
>>> quote = 'SIMPLICITY IS THE ULTIMATE SOPHISTICATION'
>>> tr_table = str.maketrans(string.ascii_uppercase, string.ascii_lowercase)
>>> quote.translate(tr_table)
'simplicity is the ultimate sophistication'

>>> sentence = "Thi1s is34 a senten6ce"
>>> sentence.translate(str.maketrans('', '', string.digits))
'This is a sentence'
>>> greeting.translate(str.maketrans('', '', string.punctuation))
' Have a great day '
```

- removing leading/trailing/both characters
- only consecutive characters from start/end string are removed
- by default whitespace characters are stripped

- if more than one character is specified, it is treated as a set and all combinations of it are used

```
>>> greeting = '      Have a nice day :)     '
>>> greeting.strip()
'Have a nice day :)'
>>> greeting.rstrip()
'      Have a nice day :)'
>>> greeting.lstrip()
'Have a nice day :)     '

>>> greeting.strip(') :')
'Have a nice day'

>>> greeting = '===== Have a great day!! ====='
>>> greeting.strip('=')
' Have a great day!! '
```

- styling
- width argument specifies total output string length

```
>>> ' Hello World '.center(40, '*')
'************ Hello World **************'
```

- changing case and case checking

```
>>> sentence = 'thIs iS a saMple StrIng'

>>> sentence.capitalize()
'This is a sample string'

>>> sentence.title()
'This Is A Sample String'

>>> sentence.lower()
'this is a sample string'

>>> sentence.upper()
'THIS IS A SAMPLE STRING'

>>> sentence.swapcase()
'THiS Is A SAmPLE sTRiNG'

>>> 'good'.islower()
True

>>> 'good'.isupper()
False
```

- check if string is made up of numbers

```
>>> '1'.isnumeric()
True
>>> 'abc1'.isnumeric()
False
>>> '1.2'.isnumeric()
False
```

- check if character sequence is present or not

```
>>> sentence = 'This is a sample string'
>>> 'is' in sentence
True
>>> 'this' in sentence
False
>>> 'This' in sentence
True
>>> 'this' in sentence.lower()
True
>>> 'is a' in sentence
True
>>> 'test' not in sentence
True
```

- get number of times character sequence is present (non-overlapping)

```
>>> sentence = 'This is a sample string'
>>> sentence.count('is')
2
>>> sentence.count('w')
0

>>> word = 'phototonic'
>>> word.count('oto')
1
```

- matching character sequence at start/end of string

```
>>> sentence
'This is a sample string'

>>> sentence.startswith('This')
True
>>> sentence.startswith('The')
False

>>> sentence.endswith('ing')
True
>>> sentence.endswith('ly')
False
```

- split string based on character sequence
- returns a list
- to split using regular expressions, use `re.split()` instead

```
>>> sentence = 'This is a sample string'

>>> sentence.split()
['This', 'is', 'a', 'sample', 'string']

>>> "oranges:5".split(':')
['oranges', '5']
>>> "oranges :: 5".split(' :: ')
['oranges', '5']

>>> "a e i o u".split(' ', maxsplit=1)
['a', 'e i o u']
>>> "a e i o u".split(' ', maxsplit=2)
['a', 'e', 'i o u']

>>> line = '{1.0 2.0 3.0}'
>>> nums = [float(s) for s in line.strip('{}').split()]
>>> nums
[1.0, 2.0, 3.0]
```

- joining list of strings

```
>>> str_list
['This', 'is', 'a', 'sample', 'string']
>>> ' '.join(str_list)
'This is a sample string'
>>> '-'.join(str_list)
'This-is-a-sample-string'

>>> c = ' :: '
>>> c.join(str_list)
'This :: is :: a :: sample :: string'
```

- replace characters
- third argument specifies how many times replace has to be performed
- variable has to be explicitly re-assigned to change its value

```
>>> phrase = '2 be or not 2 be'
>>> phrase.replace('2', 'to')
'to be or not to be'

>>> phrase
'2 be or not 2 be'

>>> phrase.replace('2', 'to', 1)
'to be or not 2 be'

>>> phrase = phrase.replace('2', 'to')
>>> phrase
'to be or not to be'
```

Further Reading

- Python docs - string methods
- python string methods tutorial

Regular Expressions

- Handy reference of regular expression (RE) elements

Meta characters	Description
`\A`	anchors matching to beginning of string
`\Z`	anchors matching to end of string
`^`	anchors matching to beginning of line
`$`	anchors matching to end of line
`.`	Match any character except newline character `\n`
|	OR operator for matching multiple patterns
`(RE)`	capturing group
`(?:RE)`	non-capturing group
`[]`	Character class - match one character among many
`\^`	prefix `\` to literally match meta characters like `^`

Greedy Quantifiers	Description
`*`	Match zero or more times
`+`	Match one or more times
`?`	Match zero or one times
`{m,n}`	Match `m` to `n` times (inclusive)
`{m,}`	Match at least m times
`{,n}`	Match up to `n` times (including `0` times)
`{n}`	Match exactly n times

Appending a `?` to greedy quantifiers makes them non-greedy

Character classes	Description
`[aeiou]`	Match any vowel
`[^aeiou]`	`^` inverts selection, so this matches any consonant
`[a-f]`	`-` defines a range, so this matches any of abcdef characters
`\d`	Match a digit, same as `[0-9]`
`\D`	Match non-digit, same as `[^0-9]` or `[^\d]`
`\w`	Match alphanumeric and underscore character, same as `[a-zA-Z0-9_]`
`\W`	Match non-alphanumeric and underscore character, same as `[^a-zA-Z0-9_]` or `[^\w]`
`\s`	Match white-space character, same as `[\ \t\n\r\f\v]`
`\S`	Match non white-space character, same as `[^\s]`
`\b`	word boundary, see `\w` for characters constituting a word
`\B`	not a word boundary

Flags	Description
`re.I`	Ignore case
`re.M`	Multiline mode, `^` and `$` anchors work on lines
`re.S`	Singleline mode, `.` will also match `\n`
`re.V`	Verbose mode, for better readability and adding comments

See Python docs - Compilation Flags for more details and long names for flags

Variable	Description
`\1` , `\2` , `\3` ... `\99`	backreferencing matched patterns
`\g<1>` , `\g<2>` , `\g<3>` ...	backreferencing matched patterns, prevents ambiguity
`\g<0>`	entire matched portion

Pattern matching and extraction

To match/extract sequence of characters, use

- `re.search()` to see if input string contains a pattern or not
- `re.findall()` to get a list of all matching patterns
- `re.split()` to get a list from splitting input string based on a pattern

Their syntax is as follows:

```
re.search(pattern, string, flags=0)
re.findall(pattern, string, flags=0)
re.split(pattern, string, maxsplit=0, flags=0)
```

- As a good practice, always use **raw strings** to construct RE, unless other formats are required
  - this will avoid clash of backslash escaping between RE and normal quoted strings
- examples for `re.search`

```
>>> sentence = 'This is a sample string'

# using normal string methods
>>> 'is' in sentence
True
>>> 'xyz' in sentence
False

# need to load the re module before use
>>> import re
# check if 'sentence' contains the pattern described by RE argument
>>> bool(re.search(r'is', sentence))
True
>>> bool(re.search(r'this', sentence, flags=re.I))
True
>>> bool(re.search(r'xyz', sentence))
False
```

- examples for `re.findall`

```
# match whole word par with optional s at start and e at end
>>> re.findall(r'\bs?pare?\b', 'par spar apparent spare part pare')
['par', 'spar', 'spare', 'pare']

# numbers >= 100 with optional leading zeros
>>> re.findall(r'\b0*[1-9]\d{2,}\b', '0501 035 154 12 26 98234')
['0501', '154', '98234']

# if multiple capturing groups are used, each element of output
# will be a tuple of strings of all the capture groups
>>> re.findall(r'(x*):(y*)', 'xx:yyy x: x:yy :y')
[('xx', 'yyy'), ('x', ''), ('x', 'yy'), ('', 'y')]

# normal capture group will hinder ability to get whole match
# non-capturing group to the rescue
>>> re.findall(r'\b\w*(?:st|in)\b', 'cost akin more east run against')
['cost', 'akin', 'east', 'against']

# useful for debugging purposes as well before applying substitution
>>> re.findall(r't.*?a', 'that is quite a fabricated tale')
['tha', 't is quite a', 'ted ta']
```

- examples for `re.split`

```
# split based on one or more digit characters
>>> re.split(r'\d+', 'Sample123string42with777numbers')
['Sample', 'string', 'with', 'numbers']

# split based on digit or whitespace characters
>>> re.split(r'[\d\s]+', '**1\f2\n3star\t7 77\r**')
['**', 'star', '**']

# to include the matching delimiter strings as well in the output
>>> re.split(r'(\d+)', 'Sample123string42with777numbers')
['Sample', '123', 'string', '42', 'with', '777', 'numbers']

# use non-capturing group if capturing is not needed
>>> re.split(r'hand(?:y|ful)', '123handed42handy777handful500')
['123handed42', '777', '500']
```

- backreferencing

```
# whole words that have at least one consecutive repeated character
>>> words = ['effort', 'flee', 'facade', 'oddball', 'rat', 'tool']

>>> [w for w in words if re.search(r'\b\w*(\w)\1\w*\b', w)]
['effort', 'flee', 'oddball', 'tool']
```

- The `re.search` function returns a `re.Match` object from which various details can be extracted like the matched portion of string, location of matched portion, etc
- **Note** that output here is shown for Python version **3.7**

```
>>> re.search(r'b.*d', 'abc ac adc abbbc')
<re.Match object; span=(1, 9), match='bc ac ad'>
# retrieving entire matched portion
>>> re.search(r'b.*d', 'abc ac adc abbbc')[0]
'bc ac ad'

# capture group example
>>> m = re.search(r'a(.*)d(.*a)', 'abc ac adc abbbc')
# to get matched portion of second capture group
>>> m[2]
'c a'
# to get a tuple of all the capture groups
>>> m.groups()
('bc ac a', 'c a')
```

Search and Replace

Syntax

```
re.sub(pattern, repl, string, count=0, flags=0)
```

- examples
- **Note** that as strings are immutable, `re.sub` will not change value of variable passed to it, has to be explicity assigned

```
>>> ip_lines = "catapults\nconcatenate\ncat"
>>> print(re.sub(r'^', r'* ', ip_lines, flags=re.M))
* catapults
* concatenate
* cat

# replace 'par' only at start of word
>>> re.sub(r'\bpar', r'X', 'par spar apparent spare part')
'X spar apparent spare Xt'

# same as: r'part|parrot|parent'
>>> re.sub(r'par(en|ro)?t', r'X', 'par part parrot parent')
'par X X X'

# remove first two columns where : is delimiter
>>> re.sub(r'\A([^:]+:){2}', r'', 'foo:123:bar:baz', count=1)
'bar:baz'
```

- backreferencing

```
# remove any number of consecutive duplicate words separated by space
# quantifiers can be applied to backreferences too!
>>> re.sub(r'\b(\w+)( \1)+\b', r'\1', 'a a a walking for for a cause')
'a walking for a cause'

# add something around the matched strings
>>> re.sub(r'\d+', r'(\g<0>0)', '52 apples and 31 mangoes')
'(520) apples and (310) mangoes'

# swap words that are separated by a comma
>>> re.sub(r'(\w+),(\w+)', r'\2,\1', 'a,b 42,24')
'b,a 24,42'
```

- using functions in replace part of `re.sub()`
- **Note** that Python version **3.7** is used here

```
>>> from math import factorial
>>> numbers = '1 2 3 4 5'
>>> def fact_num(n):
...     return str(factorial(int(n[0])))
...
>>> re.sub(r'\d+', fact_num, numbers)
'1 2 6 24 120'

# using lambda
>>> re.sub(r'\d+', lambda m: str(factorial(int(m[0]))), numbers)
'1 2 6 24 120'
```

- call functions from re.sub
- replace string pattern with output of function
- lambda tutorial

Compiling Regular Expressions

- Regular expressions can be compiled using `re.compile` function, which gives back a `re.Pattern` object
- The top level `re` module functions are all available as methods for this object
- Compiling a regular expression helps if the RE has to be used in multiple places or called upon multiple times inside a loop (speed benefit)
- By default, Python maintains a small list of recently used RE, so the speed benefit doesn't apply for trivial use cases

```
>>> pet = re.compile(r'dog')
>>> type(pet)
<class 're.Pattern'>
>>> bool(pet.search('They bought a dog'))
True
>>> bool(pet.search('A cat crossed their path'))
False

>>> remove_parentheses = re.compile(r'\([^)]*\)')
>>> remove_parentheses.sub('', 'a+b(addition) - foo() + c%d(#modulo)')
'a+b - foo + c%d'
>>> remove_parentheses.sub('', 'Hi there(greeting). Nice day(a(b)')
'Hi there. Nice day'
```

Further Reading on Regular Expressions

- Python re(gex)? - a book on regular expressions
- Python docs - re module
- Python docs - introductory tutorial to using regular expressions
- Comprehensive reference: What does this regex mean?
- rexegg - tutorials, tricks and more
- regular-expressions - tutorials and tools
- CommonRegex - collection of common regular expressions
- Practice tools
  - regex101 - visual aid and online testing tool for regular expressions, select flavor as Python before use
  - regexone - interative tutorial
  - cheatsheet - one can also learn it interactively
  - regexcrossword - practice by solving crosswords, read 'How to play' section before you start

File handling

open function

- syntax: open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

```
>>> import locale
>>> locale.getpreferredencoding()
'UTF-8'
```

- We'll be seeing these modes in this chapter
  - `r` open file for reading
  - `w` open file for writing
  - `a` open file for appending
- default is text mode, so passing 'r' and 'rt' are equivalent
  - for binary mode, it would be 'rb', 'wb' and so on
- `locale.getpreferredencoding()` gives default encoding being used
- Python docs - open
- Python docs - standard encodings

Reading files

```
#!/usr/bin/python3

# open file, read line by line and print it
filename = 'hello_world.py'
f = open(filename, 'r', encoding='ascii')

print("Contents of " + filename)
print('-' * 30)
for line in f:
    print(line, end='')

f.close()

# 'with' is a simpler alternative, automatically handles file closing
filename = 'while_loop.py'

print("\n\nContents of " + filename)
print('-' * 30)
with open(filename, 'r', encoding='ascii') as f:
    for line in f:
        print(line, end='')
```

- default encoding is usually 'UTF-8', use 'ascii' where applicable
- using `with` and file handle name as `f` is usual convention

```
$ ./file_reading.py
Contents of hello_world.py
------------------------------
#!/usr/bin/python3

print("Hello World")


Contents of while_loop.py
------------------------------
#!/usr/bin/python3

# continuously ask user input till it is a positive integer
usr_string = 'not a number'
while not usr_string.isnumeric():
    usr_string = input("Enter a positive integer: ")
```

If file doesn't exist

```
#!/usr/bin/python3

with open('xyz.py', 'r', encoding='ascii') as f:
    for line in f:
        print(line, end='')
```

- Error if file is not found

```
$ ./file_reading_error.py
Traceback (most recent call last):
  File "./file_reading_error.py", line 3, in <module>
    with open('xyz.py', 'r', encoding='ascii') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'xyz.py'
$ echo $?
1
```

- read entire file content as single string using `read()`

```
>>> f = open('hello_world.py', 'r', encoding='ascii')
>>> f
<_io.TextIOWrapper name='hello_world.py' mode='r' encoding='ascii'>
>>> print(f.read())
#!/usr/bin/python3

print("Hello World")
```

- read line by line using `readline()`

```
>>> f = open('hello_world.py', 'r', encoding='ascii')
>>> print(f.readline(), end='')
#!/usr/bin/python3
>>> print(f.readline(), end='')

>>> print(f.readline(), end='')
print("Hello World")

>>> f.close()
>>> print(f.readline(), end='')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

- read all lines of file as list using `readlines()`
  - note the plural form

```
>>> f = open('hello_world.py', 'r', encoding='ascii')
>>> all_lines = f.readlines()
>>> all_lines
['#!/usr/bin/python3\n', '\n', 'print("Hello World")\n']
```

- check if file is closed or not

```
>>> f = open('hello_world.py', 'r', encoding='ascii')

>>> f.closed
False

>>> f.close()
>>> f.closed
True
```

Writing to files

```
#!/usr/bin/python3

with open('new_file.txt', 'w', encoding='ascii') as f:
    f.write("This is a sample line of text\n")
    f.write("Yet another line\n")
```

- Use the `write()` method to print a string to files
- To add text to an existing file, use 'a' mode instead of 'w'

```
$ ./file_writing.py
$ cat new_file.txt
This is a sample line of text
Yet another line
```

Inplace editing with fileinput

```
#!/usr/bin/python3

import fileinput

with fileinput.input(inplace=True) as f:
    for line in f:
        line = line.replace('line of text', 'line')
        print(line, end='')
```

- The files to be modified are specified as Command line arguments when the program is run
- Note that `print` function has to be used instead of `f.write`
- Since the line read every iteration already has newline character, **end** is assigned empty string
- Python docs - fileinput

```
$ ./inplace_file_editing.py new_file.txt
$ cat new_file.txt
This is a sample line
Yet another line

$ # to change all files in current directory ending with .txt, use
$ ./inplace_file_editing.py *.txt

$ # stdin can also be passed as input, inplace gets disabled
$ echo 'a line of text' | ./inplace_file_editing.py
a line
```

- specifying filenames and backup extensions

```
# To specify filenames within the program itself
with fileinput.input(inplace=True, files=('file1.txt', 'file2.txt')) as f:

# To create backup of unmodified files, pass an extension to backup parameter
with fileinput.input(inplace=True, backup='.bkp') as f:
```

Command line arguments

Known number of arguments

```
#!/usr/bin/python3

import sys

if len(sys.argv) != 3:
    sys.exit("Error: Please provide exactly two numbers as arguments")
else:
    (num1, num2) = sys.argv[1:]
    total = int(num1) + int(num2)
    print("{} + {} = {}".format(num1, num2, total))
```

- Command line arguments given to Python program are automatically saved in `sys.argv` list
- It is a good idea to let the user know something went wrong
- As the program terminates with error message for wrong usage, use `sys.exit()` for error message (exit status 1) or a custom exit status number
- Python docs - sys module

```
$ ./sum_of_two_numbers.py 2 3
2 + 3 = 5
$ echo $?
0

$ ./sum_of_two_numbers.py 2 3 7
Error: Please provide exactly two numbers as arguments
$ echo $?
1

$ ./sum_of_two_numbers.py 2 'x'
Traceback (most recent call last):
  File "./sum_of_two_numbers.py", line 9, in <module>
    total = int(num1) + int(num2)
ValueError: invalid literal for int() with base 10: 'x'
$ echo $?
1
```

Varying number of arguments

```
#!/usr/bin/python3

import sys, pathlib, subprocess

if len(sys.argv) < 2:
    sys.exit("Error: Please provide atleast one filename as argument")

input_files = sys.argv[1:]
files_not_found = []

for filename in input_files:
    if not pathlib.Path(filename).is_file():
        files_not_found.append("File '{}' not found".format(filename))
        continue

    line_count = subprocess.getoutput('wc -l < ' + filename)
    print("{0:40}: {1:4} lines".format(filename, line_count))

print("\n".join(files_not_found))
```

- When dealing with filenames obtained as user input, it is good to do a sanity check before processing
- Python docs - pathlib

```
$ ./varying_command_line_args.py
Error: Please provide atleast one filename as argument
$ echo $?
1

$ #selective output presented
$ ./varying_command_line_args.py *.py
calling_shell_commands.py               : 14   lines
for_loop.py                             : 6    lines
functions_default_arg_value.py          : 16   lines
functions.py                            : 24   lines
hello_world.py                          : 3    lines
if_elif_else.py                         : 22   lines
if_else_oneliner.py                     : 6    lines
shell_command_output_redirections.py    : 21   lines
...

$ ./varying_command_line_args.py hello_world.py xyz.py for_loop.py abc.py
hello_world.py                          : 3    lines
for_loop.py                             : 6    lines
File 'xyz.py' not found
File 'abc.py' not found
```

- use `os` module instead of `pathlib` for file checking if your Python version is not 3.4 and higher

```
import os

if not os.path.isfile(filename):
    sys.exit("File '{}' not found".format(filename))
```

Using program name in code

```
#!/usr/bin/python3

import sys, pathlib, subprocess, re

if len(sys.argv) != 2:
    sys.exit("Error: Please provide exactly one filename as argument")

program_name = sys.argv[0]
filename = sys.argv[1]

if not pathlib.Path(filename).is_file():
    sys.exit("File '{}' not found".format(filename))

if re.search(r'line_count.py', program_name):
    lc = subprocess.getoutput('wc -l < ' + filename)
    print("No. of lines in '{}' is: {}".format(filename, lc))
elif re.search(r'word_count.py', program_name):
    wc = subprocess.getoutput('wc -w < ' + filename)
    print("No. of words in '{}' is: {}".format(filename, wc))
else:
    sys.exit("Program name '{}' not recognized".format(program_name))
```

- Same program can be repurposed for different functionalities based on its name

```
$ ./line_count.py if_elif_else.py
No. of lines in 'if_elif_else.py' is: 22

$ ln -s line_count.py word_count.py
$ ./word_count.py if_elif_else.py
No. of words in 'if_elif_else.py' is: 73

$ ln -s line_count.py abc.py
$ ./abc.py if_elif_else.py
Program name './abc.py' not recognized

$ wc -lw if_elif_else.py
 22  73 if_elif_else.py
```

Command line switches

```
#!/usr/bin/python3

import argparse, subprocess

parser = argparse.ArgumentParser()
parser.add_argument('-f', '--file', help="file to be sorted", required=True)
parser.add_argument('-u', help="sort uniquely", action="store_true")
args = parser.parse_args()

if args.u:
    subprocess.call(['sort', '-u', args.file, '-o', args.file])
else:
    subprocess.call(['sort', args.file, '-o', args.file])
```

- using `argparse` module is a simpler way to build programs that behave like shell commands
- By default it adds a help option `-h` (short form) and `--help` (long form) as well as handles wrong usage

In this example:

- `-f` or `--file` option is declared as required option, it accepts an argument (which we treat as input file name in code)
- `-u` is an optional flag
- the `help` argument is used to specify text to be displayed for those options in help message

```
$ ./sort_file.py
usage: sort_file.py [-h] -f FILE [-u]
sort_file.py: error: the following arguments are required: -f/--file

$ ./sort_file.py -h
usage: sort_file.py [-h] -f FILE [-u]

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  file to be sorted
  -u                    sort uniquely

$ ./sort_file.py -f test_list.txt
$ cat test_list.txt
async_test
basic_test
input_test
input_test
output_test
sync_test

$ ./sort_file.py -f test_list.txt -u
$ cat test_list.txt
async_test
basic_test
input_test
output_test
sync_test

$ ./sort_file.py -f xyz.txt
sort: cannot read: xyz.txt: No such file or directory
```

- specifying positional arguments

```
#!/usr/bin/python3

import argparse

parser = argparse.ArgumentParser()
parser.add_argument('num1', type=int, help="first number")
parser.add_argument('num2', type=int, help="second number")
args = parser.parse_args()

total = args.num1 + args.num2
print("{} + {} = {}".format(args.num1, args.num2, total))
```

- more readable approach than manually parsing `sys.argv` as well as default help and error handling
- type conversions required can be specified while building parser itself

```
$ ./sum2nums_argparse.py
usage: sum2nums_argparse.py [-h] num1 num2
sum2nums_argparse.py: error: the following arguments are required: num1, num2

$ ./sum2nums_argparse.py -h
usage: sum2nums_argparse.py [-h] num1 num2

positional arguments:
  num1        first number
  num2        second number

optional arguments:
  -h, --help  show this help message and exit

$ ./sum2nums_argparse.py 3 4
3 + 4 = 7

$ ./sum2nums_argparse.py 3 4 7
usage: sum2nums_argparse.py [-h] num1 num2
sum2nums_argparse.py: error: unrecognized arguments: 7
```

Further Reading

- Python docs - argparse tutorial
- Python docs - argparse module
- argparse examples with explanation
- Python docs - getopt module

Exception Handling and Debugging

Exception Handling

- We have seen plenty of errors in previous chapters when something goes wrong or some input was given erroneously
- For example:

```
$ ./user_input_int.py
Enter an integer number: abc
Traceback (most recent call last):
  File "./user_input_int.py", line 6, in <module>
    usr_num = int(usr_ip)
ValueError: invalid literal for int() with base 10: 'abc'
```

- In such cases, it might be preferred to inform the user on the error and give a chance to correct it
- Python provides the `try-except` construct to achieve this

```
#!/usr/bin/python3

while True:
    try:
        usr_num = int(input("Enter an integer number: "))
        break
    except ValueError:
        print("Not an integer, try again")

print("Square of entered number is: {}".format(usr_num * usr_num))
```

- `except` can be used for particular error (in this case `ValueError`)

```
$ ./user_input_exception.py
Enter an integer number: a
Not an integer, try again
Enter an integer number: 1.2
Not an integer, try again
Enter an integer number: 3
Square of entered number is: 9
```

Further Reading

- Python docs - errors, exception handling and raising exceptions
- Python docs - built-in exceptions
- stackoverflow - exception message capturing
- stackoverflow - avoid bare exceptions
- Python docs - pass statement

Syntax check

- Python's command line options can be used for variety of purposes
- Syntax checking is one of them

```
$ python3 -m py_compile syntax_error.py
  File "syntax_error.py", line 3
    print "Have a nice day"
                          ^
SyntaxError: Missing parentheses in call to 'print'
```

- Useful to quickly catch syntax errors like missing `:` for `if for with` etc and `()` for `print` statements
- While this example might be trivial, real world program might have thousands of lines and tougher to find typos
- Python docs - cmdline
  - One-liners: #1, #2, #3

pdb

- Invoking debugger is another use of `cmdline`
- Use it instead of using `print` all over a program when something goes wrong, plus one can use breakpoints and other features specific to debugging programs

```
$ python3 -m pdb if_elif_else.py
> /home/learnbyexample/python_programs/if_elif_else.py(3)<module>()
-> num = 45
(Pdb) p num
*** NameError: name 'num' is not defined
(Pdb) n
> /home/learnbyexample/python_programs/if_elif_else.py(6)<module>()
-> if num > 25:
(Pdb) p num
45
(Pdb) l
  1      #!/usr/bin/python3
  2
  3      num = 45
  4
  5      # only if
  6  ->    if num > 25:
  7            print("Hurray! {} is greater than 25".format(num))
  8
  9      # if-else
 10      if num % 2 == 0:
 11          print("{} is an even number".format(num))
(Pdb) q
```

- `l` prints code around the current statement the debugger is at, useful to visualize the progress of debug effort
- `s` execute current line, steps inside function calls
- `n` execute current line and treats function as single execution step
- `c` continue execution until next breakpoint
- `p variable` print value of variable
- `h` list of commands
  - `h c` help on `c` command
- `q` quit the debugger

Further Reading

- Python docs - pdb
- pdb tutorial
- common runtime errors
- common beginner errors as a flowchart
- Common pitfalls
- Python docs - Basic Logging Tutorial

Importing program

- One can also `import` a program directly in Interpreter to test functions
- `if __name__ == "__main__":` construct
  - code inside that construct will be executed when program is intended to be run - ex: executing the program from command line
  - code inside that construct will NOT be executed when program is intended to be imported as module - ex: to use programs's functions
- A good practice is to put all code outside of functions inside a `main` function and call `main()` inside the `if __name__ == "__main__":` construct
- Note that `__name__` and `__main__` have two underscore characters as prefix and suffix

```
#!/usr/bin/python3

# ----- function without arguments -----
def greeting():
    print("-----------------------------")
    print("         Hello World         ")
    print("-----------------------------")

# ----- function with arguments -----
def sum_two_numbers(num1, num2):
    sum = num1 + num2
    print("{} + {} = {}".format(num1, num2, sum))

# ----- function with return value -----
def num_square(num):
    return num * num

# ----- main -----
def main():
    greeting()
    sum_two_numbers(3, 4)

    my_num = 3
    print(num_square(2))
    print(num_square(my_num))

if __name__ == "__main__":
    main()
```

- When run as a program

```
$ ./functions_main.py
----------------------------
         Hello World
----------------------------
3 + 4 = 7
4
9
```

- When importing

```
>>> import functions_main

>>> functions_main.greeting()
----------------------------
         Hello World
----------------------------

>>> functions_main.num_square()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: num_square() missing 1 required positional argument: 'num'

>>> functions_main.num_square(5)
25

>>> functions_main.main()
----------------------------
         Hello World
----------------------------
3 + 4 = 7
4
9
```

Further Reading

- Python docs - __main__
- What does if __name__ == "__main__" do?
- Python packaging guide
- diveintopython3 - packaging

Docstrings

Style guide

Paraphrased from Python docs - coding style

- Use 4-space indentation, and no tabs.
  - 4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.
- Wrap lines so that they don't exceed 79 characters.
  - This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas
- Name your classes and functions consistently;
  - the convention is to use CamelCase for classes and lower_case_with_underscores for functions and methods

Style guides

- PEP 0008
- Google - pyguide
- elements of python style
- The Hitchhiker's Guide to Python - handbook of best practices

Palindrome example

```
#!/usr/bin/python3

"""
Asks for user input and tells if string is palindrome or not

Allowed characters: alphabets and punctuations .,;:'"-!?
Minimum alphabets: 3 and cannot be all same

Informs if input is invalid and asks user for input again
"""

import re

def is_palindrome(usr_ip):
    """
    Checks if string is a palindrome

    ValueError: if string is invalid

    Returns True if palindrome, False otherwise
    """

    # remove punctuations & whitespace and change to all lowercase
    ip_str = re.sub(r'[\s.;:,\'"!?-]', r'', usr_ip).lower()

    if re.search(r'[^a-zA-Z]', ip_str):
        raise ValueError("Characters other than alphabets and punctuations")
    elif len(ip_str) < 3:
        raise ValueError("Less than 3 alphabets")
    else:
        return ip_str == ip_str[::-1] and not re.search(r'^(.)\1+$', ip_str)

def main():
    while True:
        try:
            usr_ip = input("Enter a palindrome: ")
            if is_palindrome(usr_ip):
                print("{} is a palindrome".format(usr_ip))
            else:
                print("{} is NOT a palindrome".format(usr_ip))
            break
        except ValueError as e:
            print('Error: ' + str(e))

if __name__ == "__main__":
    main()
```

- The first triple quoted strings marks the docstring for entire program
- The second one inside `is_palindrome()` is specific for that function

```
$ ./palindrome.py
Enter a palindrome: as2
Error: Characters other than alphabets and punctuations
Enter a palindrome: "Dammit, I'm mad!"
"Dammit, I'm mad!" is a palindrome

$ ./palindrome.py
Enter a palindrome: a'a
Error: Less than 3 alphabets
Enter a palindrome: aaa
aaa is NOT a palindrome
```

- Let's see how docstrings can be used as help
- Notice how docstring gets automatically formatted

```
>>> import palindrome
>>> help(palindrome)

Help on module palindrome:

NAME
    palindrome - Asks for user input and tells if string is palindrome or not

DESCRIPTION
    Allowed characters: alphabets and punctuations .,;:'"-!?
    Minimum alphabets: 3 and cannot be all same

    Informs if input is invalid and asks user for input again

FUNCTIONS
    is_palindrome(usr_ip)
        Checks if string is a palindrome

        ValueError: if string is invalid

        Returns True if palindrome, False otherwise

    main()

FILE
    /home/learnbyexample/python_programs/palindrome.py
```

- One can get help on functions directly too

```
>>> help(palindrome.is_palindrome)

Help on function is_palindrome in module palindrome:

is_palindrome(usr_ip)
    Checks if string is a palindrome

    ValueError: if string is invalid

    Returns True if palindrome, False otherwise
```

- and of course test the functions

```
>>> palindrome.is_palindrome('aaa')
False
>>> palindrome.is_palindrome('Madam')
True

>>> palindrome.main()
Enter a palindrome: 3452
Error: Characters other than alphabets and punctuations
Enter a palindrome: Malayalam
Malayalam is a palindrome
```

Further Reading

- docstring formats
- exception message capturing

Testing

assert statement

- `assert` is primarily used for debugging purposes like catching invalid input or a condition that shouldn't occur
- An optional message can be passed for descriptive error message than a plain **AssertionError**
- It uses raise statement for implementation
- `assert` statements can be skipped by passing the `-O` command line option
- **Note** that `assert` is a statement and not a function

```
>>> assert 2 ** 3 == 8
>>> assert 3 > 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError

>>> assert 3 > 4, "3 is not greater than 4"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: 3 is not greater than 4
```

Let's take a factorial function as an example:

```
>>> def fact(n):
        total = 1
        for num in range(1, n+1):
            total *= num
        return total

>>> assert fact(4) == 24
>>> assert fact(0) == 1
>>> fact(5)
120

>>> fact(-3)
1
>>> fact(2.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in fact
TypeError: 'float' object cannot be interpreted as an integer
```

- `assert fact(4) == 24` and `assert fact(0) == 1` can be considered as sample tests to check the function

Let's see how `assert` can be used to validate arguments passed to the function:

```
>>> def fact(n):
        assert type(n) == int and n >= 0, "Number should be zero or positive integer"

        total = 1
        for num in range(1, n+1):
            total *= num
        return total

>>> fact(5)
120
>>> fact(-3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fact
AssertionError: Number should be zero or positive integer
>>> fact(2.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fact
AssertionError: Number should be zero or positive integer
```

The above factorial function can also be written using reduce

```
>>> def fact(n):
        assert type(n) == int and n >= 0, "Number should be zero or positive integer"

        from functools import reduce
        from operator import mul
        return reduce(mul, range(1, n+1), 1)


>>> fact(23)
25852016738884976640000
```

Above examples for demonstration only, for practical purposes use `math.factorial` which also gives appropriate exceptions

```
>>> from math import factorial
>>> factorial(10)
3628800

>>> factorial(-5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: factorial() not defined for negative values
>>> factorial(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: factorial() only accepts integral values
```

Further Reading

- Python docs - assert
- What is the use of assert in Python?
- Is Unit Testing worth the effort?

Using assert to test a program

In a limited fashion, one can use `assert` to test a program - either within the program (and later skipped using the `-O` option) or as separate test program(s)

Let's try testing the **palindrome** program we saw in Docstrings chapter

```
#!/usr/bin/python3

import palindrome

assert palindrome.is_palindrome('Madam')
assert palindrome.is_palindrome("Dammit, I'm mad!")
assert not palindrome.is_palindrome('aaa')
assert palindrome.is_palindrome('Malayalam')

try:
    assert palindrome.is_palindrome('as2')
except ValueError as e:
    assert str(e) == 'Characters other than alphabets and punctuations'

try:
    assert palindrome.is_palindrome("a'a")
except ValueError as e:
    assert str(e) == 'Less than 3 alphabets'

print('All tests passed')
```

- There are four different cases tested for **is_palindrome** function
  - Valid palindrome string
  - Invalid palindrome string
  - Invalid characters in string
  - Insufficient characters in string
- Both the program being tested and program to test are in same directory
- To test the **main** function, we need to simulate user input. For this and other useful features, test frameworks come in handy

```
$ ./test_palindrome.py
All tests passed
```

Using unittest framework

This section requires understanding of classes

```
#!/usr/bin/python3

import palindrome
import unittest

class TestPalindrome(unittest.TestCase):

    def test_valid(self):
        # check valid input strings
        self.assertTrue(palindrome.is_palindrome('kek'))
        self.assertTrue(palindrome.is_palindrome("Dammit, I'm mad!"))
        self.assertFalse(palindrome.is_palindrome('zzz'))
        self.assertFalse(palindrome.is_palindrome('cool'))

    def test_error(self):
        # check only the exception raised
        with self.assertRaises(ValueError):
            palindrome.is_palindrome('abc123')

        with self.assertRaises(TypeError):
            palindrome.is_palindrome(7)

        # check error message as well
        with self.assertRaises(ValueError) as cm:
            palindrome.is_palindrome('on 2 no')
        em = str(cm.exception)
        self.assertEqual(em, 'Characters other than alphabets and punctuations')

        with self.assertRaises(ValueError) as cm:
            palindrome.is_palindrome('to')
        em = str(cm.exception)
        self.assertEqual(em, 'Less than 3 alphabets')

if __name__ == '__main__':
    unittest.main()
```

- First we create a subclass of **unittest.TestCase** (inheritance)
- Then, different type of checks can be grouped in separate functions - function names starting with **test** are automatically called by **unittest.main()**
- Depending upon type of test performed, **assertTrue, assertFalse, assertRaises, assertEqual, etc** can be used
- An Introduction to Classes and Inheritance
- Python docs - unittest
  - Command-Line Interface
  - Test Discovery

- Organizing test code, setUp and tearDown

```
$ ./unittest_palindrome.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK

$ ./unittest_palindrome.py -v
test_error (__main__.TestPalindrome) ... ok
test_valid (__main__.TestPalindrome) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Using unittest.mock to test user input and program output

This section requires understanding of decorators, do check out this wonderful intro

A simple example to see how to capture `print` output for testing

```
>>> from unittest import mock
>>> from io import StringIO

>>> def greeting():
        print('Hi there!')

>>> def test():
        with mock.patch('sys.stdout', new_callable=StringIO) as mock_stdout:
            greeting()
            assert mock_stdout.getvalue() == 'Hi there!\n'

>>> test()
```

One can also use `decorators`

```
>>> @mock.patch('sys.stdout', new_callable=StringIO)
    def test(mock_stdout):
        greeting()
        assert mock_stdout.getvalue() == 'Hi there!\n'
```

Now let's see how to emulate `input`

```
>>> def greeting():
        name = input('Enter your name: ')
        print('Hello', name)

>>> greeting()
Enter your name: learnbyexample
Hello learnbyexample

>>> with mock.patch('builtins.input', return_value='Tom'):
        greeting()

Hello Tom
```

Combining both

```
>>> @mock.patch('sys.stdout', new_callable=StringIO)
    def test_greeting(name, mock_stdout):
        with mock.patch('builtins.input', return_value=name):
            greeting()
            assert mock_stdout.getvalue() == 'Hello ' + name + '\n'

>>> test_greeting('Jo')
```

Having seen basic input/output testing, let's apply it to main function of **palindrome**

```
#!/usr/bin/python3

import palindrome
import unittest
from unittest import mock
from io import StringIO

class TestPalindrome(unittest.TestCase):

    @mock.patch('sys.stdout', new_callable=StringIO)
    def main_op(self, tst_str, mock_stdout):
        with mock.patch('builtins.input', side_effect=tst_str):
            palindrome.main()
        return mock_stdout.getvalue()

    def test_valid(self):
        for s in ('Malayalam', 'kek'):
            self.assertEqual(self.main_op([s]), s + ' is a palindrome\n')

        for s in ('zzz', 'cool'):
            self.assertEqual(self.main_op([s]), s + ' is NOT a palindrome\n')

    def test_error(self):
        em1 = 'Error: Characters other than alphabets and punctuations\n'
        em2 = 'Error: Less than 3 alphabets\n'

        tst1 = em1 + 'Madam is a palindrome\n'
        self.assertEqual(self.main_op(['123', 'Madam']), tst1)

        tst2 = em2 + em1 + 'Jerry is NOT a palindrome\n'
        self.assertEqual(self.main_op(['to', 'a2a', 'Jerry']), tst2)

if __name__ == '__main__':
    unittest.main()
```

- Two test functions - one for testing valid input strings and another to check error messages
- Here, **side_effect** which accepts iterable like list, compared to **return_value** where only one input value can be mocked
- For valid input strings, the **palindrome** main function would need only one input value
- For error conditions, the iterable comes handy as the main function is programmed to run infinitely until valid input is given
- Python docs - unittest.mock
  - patchers
- An Introduction to Mocking in Python
- PEP 0318 - decorators

- decorators

```
$ ./unittest_palindrome_main.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

Other testing frameworks

- pytest
- Python docs - doctest
- Python test automation
- Python Testing Tools Taxonomy
- Python test frameworks

Test driven development (TDD)

- Test-Driven Development with Python
- learn Python via TDD

Exercises

1) Variables and Print 2) Functions 3) Control structures 4) List 5) File 6) Text processing 7) Misc

For some questions, Python program with assert statements is provided to automatically test your solution in the exercise_files directory - for ex: Q2a - length of integer numbers. The directory also has sample input text files.

You can also solve these exercises on repl.it, with an option to submit them for review.

1) Variables and Print

Q1a) Ask user information, for ex: `name` , `department` , `college` etc and display them using print function

```
# Sample of how program might ask user input and display output afterwards
$ ./usr_ip.py
Please provide the following details
Enter your name: learnbyexample
Enter your department: ECE
Enter your college: PSG Tech

-----------------------------------
Name       : learnbyexample
Department : ECE
College    : PSG Tech
```

2) Functions

Q2a) Returns length of integer numbers

```
>>> len_int(962306349871524124750813401378124)
33
>>> len_int(+42)
2
>>> len_int(-42)
3

# bonus: handle -ve numbers and check for input type
>>> len_int(-42)
2
# len_int('a') should give
TypeError: provide only integer input
```

Q2b) Returns True/False - two strings are same irrespective of lowercase/uppercase

```
>>> str_cmp('nice', 'nice')
True
>>> str_cmp('Hi there', 'hi there')
True
>>> str_cmp('GoOd DaY', 'gOOd dAy')
True
>>> str_cmp('foo', 'food')
False
```

Q2c) Returns True/False - two strings are anagrams (assume input consists of alphabets only)

```
>>> str_anagram('god', 'Dog')
True
>>> str_anagram('beat', 'table')
False
>>> str_anagram('Tap', 'paT')
True
>>> str_anagram('beat', 'abet')
True
```

Q2d) Returns corresponding integer or floating-point number (See Number and String data types chapter for details)

```
# number input
>>> num(3)
3
>>> num(0x1f)
31
>>> num(3.32)
3.32

# string input
>>> num('123')
123
>>> num('-78')
-78
>>> num(" 42  \n ")
42
>>> num('3.14')
3.14
>>> num('3.982e5')
398200.0

>>> s = '56'
>>> num(s) + 44
100
```

Other than integer or floating, only string data type should be accepted. Also, provide custom error message if input cannot be converted

```
# num(['1', '2.3'])
TypeError: not a valid input

# num('foo')
ValueError: could not convert string to int or float
```

3) Control structures

Q3a) Write a function that returns

- 'Good' for numbers divisible by 7
- 'Food' for numbers divisible by 6
- 'Universe' for numbers divisible by 42
- 'Oops' for all other numbers
- Only one output, divisible by 42 takes precedence

```
>>> six_by_seven(66)
'Food'
>>> six_by_seven(13)
'Oops'
>>> six_by_seven(42)
'Universe'
>>> six_by_seven(14)
'Good'
>>> six_by_seven(84)
'Universe'
>>> six_by_seven(235432)
'Oops'
```

bonus: use a loop to print number and corresponding string for numbers 1 to 100

```
1 Oops
2 Oops
3 Oops
4 Oops
5 Oops
6 Food
7 Good
...
41 Oops
42 Universe
...
98 Good
99 Oops
100 Oops
```

Q3b) Print all numbers from 1 to 1000 which reads the same in reversed form in both binary and decimal format

```
$ ./dec_bin.py
1 1
3 11
5 101
7 111
9 1001
33 100001
99 1100011
313 100111001
585 1001001001
717 1011001101
```

bonus: add octal and hexadecimal checks as well

```
$ ./dec_bin_oct.py
1 0b1 0o1
3 0b11 0o3
5 0b101 0o5
7 0b111 0o7
9 0b1001 0o11
585 0b1001001001 0o1111

$ ./dec_bin_oct_hex.py
1 0b1 0o1 0x1
3 0b11 0o3 0x3
5 0b101 0o5 0x5
7 0b111 0o7 0x7
9 0b1001 0o11 0x9
```

4) List

Q4a) Write a function that returns product of all numbers of a list

```
>>> product([1, 4, 21])
84
>>> product([-4, 2.3e12, 77.23, 982, 0b101])
-3.48863356e+18
```

bonus: works on any kind of iterable

```
>>> product((-3, 11, 2))
-66
>>> product({8, 300})
2400
>>> product([234, 121, 23, 945, 0])
0
>>> product(range(2, 6))
120
# can you identify what mathematical function the last one performs?
```

Q4b) Write a function that returns nth lowest number of a list (or iterable in general). Return the lowest if second argument not specified

Note that if a list contains duplicates, they should be handled before determining nth lowest

```
>>> nums = [42, 23421341, 234.2e3, 21, 232, 12312, -2343]
>>> nth_lowest(nums, 3)
42
>>> nth_lowest(nums, 5)
12312
>>> nth_lowest(nums, 15)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in nth_lowest
IndexError: list index out of range

>>> nums = [1, -2, 4, 2, 1, 3, 3, 5]
>>> nth_lowest(nums)
-2
>>> nth_lowest(nums, 4)
3
>>> nth_lowest(nums, 5)
4

>>> nth_lowest('unrecognizable', 3)
'c'
>>> nth_lowest('abracadabra', 4)
'd'
```

Q4c) Write a function that accepts a string input and returns slices

- if input string is less than 3 characters long, return a list with input string as the only element
- otherwise, return list with all string slices greater than 1 character long
- order of slices should be same as shown in examples below

```
>>> word_slices('i')
['i']
>>> word_slices('to')
['to']

>>> word_slices('are')
['ar', 'are', 're']
>>> word_slices('table')
['ta', 'tab', 'tabl', 'table', 'ab', 'abl', 'able', 'bl', 'ble', 'le']
```

5) File

Q5a) Print sum of all numbers from a file containing only single column and all numbers

```
$ cat f1.txt
8
53
3.14
84
73e2
100
2937

$ ./col_sum.py
10485.14
```

Q5b) Print sum of all numbers (assume only positive integer numbers) from a file containing arbitrary string

```
$ cat f2.txt
Hello123 World 35
341 2
Good 13day
How are 1784 you

$ ./extract_sum.py
2298
```

Q5c) Sort file contents in alphabetic order based on each line's extension

- extension here is defined as the string after the last `.` in the line
- if line doesn't have a `.`, those lines should come before lines with `.`
- sorting should be case-insensitive
- use rest of string as tie-breaker if there are more than one line with same extension
- assume input file is ASCII encoded and small enough to fit in memory

bonus: instead of printing results to stdout, change the input file itself with sorted result

```
$ cat f3.txt
power.Log
foo.123.txt
list
report_12.log
baz.TXT
hello.RB
loop.do.rb
Fav_books

$ ./sort_by_ext.py
Fav_books
list
power.Log
report_12.log
hello.RB
loop.do.rb
baz.TXT
foo.123.txt
```

6) Text processing

Q6a) Check if two words are same or differ by only one character (irrespective of case), input strings should have same length

See also Levenshtein distance

```
>>> is_one_char_diff('bar', 'bar')
True
>>> is_one_char_diff('bar', 'Baz')
True
>>> is_one_char_diff('Food', 'fold')
True
>>> is_one_char_diff('A', 'b')
True

>>> is_one_char_diff('a', '')
False
>>> is_one_char_diff('Bar', 'Bark')
False
>>> is_one_char_diff('Bar', 'art')
False
>>> is_one_char_diff('Food', 'fled')
False
>>> is_one_char_diff('ab', '')
False
```

Q6b) Check if a word is in ascending/descending alphabetic order or not (irrespective of case)

Can you think of a way to do it only using built-in functions and string methods?

```
>>> is_alpha_order('bot')
True
>>> is_alpha_order('arT')
True
>>> is_alpha_order('are')
False
>>> is_alpha_order('boat')
False

>>> is_alpha_order('toe')
True
>>> is_alpha_order('Flee')
False
>>> is_alpha_order('reed')
True
```

bonus: Check if all words in a sentence (assume only whitespace separated input) are in ascending/descending alphabetic order (irrespective of case)

hint use a built-in function and generator expression

```
>>> is_alpha_order_sentence('Toe got bit')
True
>>> is_alpha_order_sentence('All is well')
False
```

Q6c) Find the maximum nested depth of curly braces

Unbalanced or wrongly ordered braces should return `-1`

Iterating over input string is one way to solve this, another is to use regular expressions

```
>>> max_nested_braces('a*b')
0
>>> max_nested_braces('{a+2}*{b+c}')
1
>>> max_nested_braces('{{a+2}*{{b+{c*d}}+e*d}}')
4
>>> max_nested_braces('a*b+{}')
1
>>> max_nested_braces('}a+b{')
-1
>>> max_nested_braces('a*b{')
-1
```

bonus: empty braces, i.e `{}` should return `-1`

```
>>> max_nested_braces('a*b+{}')
-1
>>> max_nested_braces('a*{b+{}+c*{e*3.14}}')
-1
```

7) Misc

Q7a) Play a song (**hint** use `subprocess` module)

Q7b) Open a browser along with any link, for ex: https://github.com/learnbyexample/Python_Basics (**hint** use `webbrowser` module)

Q7c) Write a function that

- accepts a filesystem path(default) or a url(indicated by True as second argument)
- returns the longest word(here word is defined as one or more consecutive sequence of alphabets of either case)

- assume that input encoding is **utf-8** and small enough to fit in memory and that there's only one distinct longest word

```
>>> ip_path = 'poem.txt'
>>> longest_word(ip_path)
'Violets'

>>> ip_path = 'https://www.gutenberg.org/files/60/60.txt'
>>> longest_word(ip_path, True)
'misunderstandings'
```

For reference solutions, see exercise_solutions directory

Further Reading

Python curated resources - online courses, books, coding practice sites, frameworks for GUI/Game/Web, cheatsheet, online help forums, etc..

Standard topics not covered

- Python docs - classes
- Python docs - virtual environment
- Python docs - modules and packaging
- Python packaging guide

Useful links on coding

- Python docs - FAQ
- list of common questions on stackoverflow
- profiling Python code for performance
- pylint
- code patterns
- Performance tips

Python extensions

- Cython - optimising static compiler for both the Python programming language and the extended Cython programming language
- Jython - Python for the Java Platform
- NumPy - fundamental package for scientific computing with Python