1. Introduction

- `Python` is a general purpose, high level language.
- It is dynamically typed and interpretered language.

1.1 Getting Started

- Python runs mostly on all modern operating systems.
- Windows - http://docs.python-guide.org/en/latest/starting/install/win/ (http://docs.python-guide.org/en/latest/starting/install/win/)
- GNU/Linux - http://docs.python-guide.org/en/latest/starting/install/linux/ (http://docs.python-guide.org/en/latest/starting/install/linux/)
- Mac OSX - http://docs.python-guide.org/en/latest/starting/install/osx/ (http://docs.python-guide.org/en/latest/starting/install/osx/)

1.2 Python Interpreter

- Type `python` in the `Command Prompt` or `Terminal`. The output should look like

```
➜  learnpythoninminutes  python
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- `Command` takes you to `Python Interactive Shell`. The important thing to note here is `Python 2.7.5` which says `python version` installed in the machine. All the tutorial will focus on `Python 2.7.x`.

1.3 Using Python Interpreter

```
>>> print("Welcome to learn python in minutes tutorial")
Welcome to learn python in minutes tutorial
>>> place = "Bangalore"
>>> duration = 180 # Duration in minutes. This is a comment!
>>> author = "Kracekumar"
>>> print("Welcome to learn python in minutes tutorial by %s for %d minutes in
%s" % (author, duration, place))
Welcome to learn python in minutes tutorial by Kracekumar for 180 minutes in
Bangalore
```

- >>> indicates we are inside Python interpreter.
- Interpreter reads everyline, evaluates and returns result if any.
- `place = "Bangalore"` is variable. Left hand side of = is the name of the variable and right hand side is the value.

- No need to give what is the type of the variable.
- `print()` is function which prints result to `standard output`, here it is Python Interpreter.
- `duration = 180` is a variable which stores `int` value.
- `%s, %d` are format specifiers like `C`.

1.4 Numbers

- `Python` has very good support for number crunching.

```
>>> duration_in_mins = 180
>>> duration_in_mins / 60 # Hours, in int
3
>>> duration_in_mins / 60.0 # Hours, in float
3.0
>>> duration_in_mins * 60 # In seconds
10800
>>> 12 + 23 # Simple add
35
>>> 12 - 23 # Subtraction
-11
>>> 12 / 23 # Division
0
>>> 12 / 23.0 # Note the denominator
0.5217391304347826
>>> 12 * 23.0 # multiply
276.0
>>> 3 % 2 # Modulo
1
>>> 12 ** 23.0 # 12 power of 23
6.624737266949237e+24 # When number is too large scientific notation is used
>>> 12 ** 12
8916100448256
>>> 12 ** 18
26623333280885243904L # ‘L‘ at the end says it is a long number.
>>> 12 + 20 * 2 - 4 * 2 / 2.0
48.0
>>> 12 + (20 * 2) - 4 * (2 / 2.0)
48.0
>>> 12 + (20 * 2) - (4 * 2 ) / 2.0
48.0
```

- While dividing two numbers if both are `int` result will be `int` else `float`.

Let's see how expression are handled without brackets.

```
12 + 20 * 2 - 4 * 2 / 2.0
    12 + 40 - 4 * 2 / 2.0
        12 + 40 - 8 / 2.0
            12 + 40 - 4.0
                52 - 4.0
```

```
                    48.0
```

- Expression is evaluated from `Left -> Right`. `BODMAS (Bracket Of Division Multiplication)` rule is followed while evaluating expression.

1.5 More about variables

- It is possible to create multiple variables in same line.

```
>>> a, b = 12, 23 # Multiple declartion
>>> print(a, b)
(12, 23)
>>> b, a = a, b
>>> print a, b # print is a statement
23 12
```

- Woot! We swapped two numbers without intermediate variable.

1.6 Summary

- `a = 23` creates a new variable, no need to mention value `type`.
- No need to use `bracket` while evaluating expression unless explict perference is needed.
- Python automatically rounds off number during division if both the numbers are `int`.

2. String

- `single quote ('python') or double quote ("python") or triple single/double quotes` can be used to create a new string.
- `strings` are immutable data type.

2.1 String Interpolation

Code

```
author = 'Kracekumar'
one_liner_about_python = "Python is simple and powerful language"
description = """Python is a general purpose, high level language.
It is dynamically typed and interpretered language.
"""

print(author)
print(one_liner_about_python)
print(description)

complete_msg = """Author: {}
One liner about Python:
- {}
Long Description:
{}""".format(author, one_liner_about_python, description)
print(complete_msg)
```

Output

```
Kracekumar
Python is simple and powerful language
Python is a general purpose, high level language.
It is dynamically typed and interpretered language.

Author: Kracekumar
One liner about Python:
- Python is simple and powerful language
Long Description:
Python is a general purpose, high level language.
It is dynamically typed and interpretered language.
```

2.2 Methods

- Everything is an object in Python.
- `string` has lot of methods like `upper`, `lower`, `strip` etc ..

Code

```
print("lower".upper())
print("upper".lower())
print("captialize".capitalize())
print(" extra spaces ".strip()) # Very useful in web development while cleaning us
er input
print("find a word's position in the sentence".find('word')) # returns starting po
sition
print("how many times the letter `e` is present in the sentence".count('e'))
print("Replace delete with remove".replace('delete', 'remove'))
print("Python is simple and powerful language".startswith('Python'))
```

Output

```
LOWER
upper
Captialize
extra spaces
7
11
Replace remove with remove
True
```

2.3 Accessing characters in string

Code

```
language = 'python'
print(language[0]) # Print first character
print(language[-1]) # print last character
print(len(language)) # builtin function, Find the length of the string
print(language[0:2]) # Print first 3 characters. Slicing `0` is starting index and
```

```
  '2' is last index.
print(language[-3:]) # print last three characters
language = language + "." # # Add a dot after python. Here new variable is created
print(language)
```

Output

```
p
n
6
py
hon
python.
```

2.4 Summary

- `string` is immutable datatype.
- `string` has lot of useful methods for data processing.
- `negative index` can be used for accessing the string content from reverse direction.
- `slicing` is used for getting partial content from the string.

3. Condition

- `True` and `False` are Boolean values in Python. `None` is used to represent absence of a value.
- `Python` uses `if`, `elif`, `else` for branching.
- `==` is used to check two values are equal, `!=` is used for non equality checks.

3.1 Boolean values

Code

```
print(23 == 32)
print(23 != 32)
print(True == False)
print(True != False)
```

Output

```
False
True
False
True
```

3.2 Branching

Write a program if a given number is divisible by `3` print `fizz`, `5` print `buzz`, divisible by both `3 and 5` print `fizz buzz` else print `nothing`. Create a new file called `fizzbuzz.py` and write the code. `python fizzbuzz.py` should execute the code.

Code

```
# fizzbuzz.py

number = 23

print(number)
if number % 15 == 0:
    print("fizz buzz")
elif number % 3 == 0:
    print("fizz")
elif number % 5 == 0:
    print("buzz")
else:
    print("nothing")
```

Output

```
examples $ python fizzbuzz.py
23
nothing
```

3.3 Few things to note

- `Python` uses indentation to separate block. So use `4 spaces` or `tabs which converts to 4 spaces`. Don't mix both.
- `:` at the end of `if`, `elif`, `else` statement.

3.4 Boolean operations

- `Python` uses keyword `and`, `or`, `not` for `and`, `or`, `not` operations.

Code

```
print(True and False)
print(23 and 12) # Watch for output
print(True or False)
print(23 or 12) # Watch for output
print("python" or "javascript")
print("python" and "javascript")
print(not 23)
```

Output

```
False
12
True
23
python
javascript
False
```

3.5 Summary

- `True`, `False` are `Boolean` values. `None` represents absence of value.
- Logical operators are `and`, `or`, `not`. Logic operators can be used with any data type.
- `==`, `!=` are comparision operators.

4 List - Data Structure

- `List` is collection of heterogenous data type.
- `List` is similar to array in other languages.
- Size of the list grows over the course of the program.

4.1 List in Action

Code

```
collection = ['Ruby', 23, 45.9, True] # Collection of different elements.
"""Representaion of list
       ---------------------------
       |'Ruby'| 23 | 45.9 | True|
       ---------------------------
        0        1      2      3

"""
print(collection)
# Access the first element
print(collection[0])
# Access the last element
print(collection[-1])
# Replace the first element
collection[0] = 'Python'
print(collection[0])
# Add an element at the last position
collection.append("last")
print(collection[-1])
# Insert an element at position 2
print(collection)
collection.insert(2, 12)
print(collection)
# Delete the last element
del collection[-1]
print(collection)
# Length of the list
print(len(collection))
```

Output

```
['Ruby', 23, 45.9, True]
Python
True
Python
```

```
last
['Python', 23, 45.9, True, 'last']
['Python', 23, 12, 45.9, True, 'last']
['Python', 23, 12, 45.9, True]
5
```

4.2 Nested list

- `List` can have list inside it.

Code

```
nested_collection = [['apple', 'orange'], ['Python', 'Go']]
print(nested_collection) # Access first element in first list
print(nested_collection[0][0])
```

Output

```
[['apple', 'orange'], ['Python', 'Go']]
apple
```

4.3 Methods

Code

```
collection = ['Python', 23, 45.9, True]
print(collection.count(23))
print(collection.index(23)) # If not found Exception will be raised.
print('23' in collection) # Check if an element is present in list
print('p' in 'python') # 'in' works on strings, list etc..
```

Output

```
1
1
False
True
```

4.4 Summary

- `List` is collection of hetergenous data types.
- `List` can have nested elements.
- `append` adds the element at the end of the list.
- `in` operator is used to check presence of an element.

5.1 For loop

- `for` loop can be used against data structure which is iterable like `list`, `string`.

5.2 loop over list

Code

```
for i in ['Python', 23, 45.9, True]: # During every iteration i value is replaced,
 ‘python‘ -> ‘23‘ -> ‘45.9‘ -> ‘True‘
    print(i)
```

Output

```
Python
23
45.9
True
```

Code

```
for ch in "Python":
    print(ch)
```

Output

```
P
y
t
h
o
n
```

Code

```
for number in range(0, 10): # Produces list of numbers starting from 0 til 9, 10 i
s excluded.
    if number % 2 == 0:
        print("Even")
    else:
        print("Odd")
```

```
Even
Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
```

5.3 Summary

- During every iteration temporary variable holds the value in the iterable.
- `:` at the end of the `for` statement is important.
- Similar to `if` statement body of `for` loop is indented.

6.1 Functions

- `Function` is unit which may take arguments, compute and returns a value. Sometimes `function` doesn't return a value.

6.2 Let's calculate

Code

```
def square(x): # No need to specify type of the argument.
    return x * x

def cube(x):
    return square(x) * x

def msg(): # Function with no arguments and doesn't return any value
    print("End")


print(square(2))
print(square(23))
msg()
```

Output

```
4
529
End
```

6.3 Default value

Code

```
def square(x=2): # if x didn't receive any value, 2 is taken.
    return x * x

print(square(23))
print(square())
```

Output

```
529
4
```

6.4 Function as function argument

- `Function` can take function as an argument.

Code

```
def fxy(f, x, y): # f(x, y) = f(x) . f(y)
    return f(x) * f(y)
```

```
def square(x):
    return x * x

print(fxy(square, 2, 3))
```

Output

```
36
```

6.5 Summary

- `Function` is small unit of computation which may take arguments and may return values.
- `Function` can be passed to a function as an argument.
- `Function` body should be indented like `if` statement.

7.1 Builtin functions

- `Python` has lot of builtin functions.

7.2 Handy builtin functions

Code

```
print(sum([1, 2, 3, 4])) # 10
print(max([1, 2, 3, 4])) # 4
print(min([1, 2, 3, 4])) # 1
print(sorted([1, 2, 3, 4])) # Ascending order
print(sorted([1, 2, 3, 4], reverse=True)) # Descending order
print(float(23)) # Typecast into float
print(list("23")) # String -> list
print(str(23))
print(int('23'))
# Find data type
print(type(23))
print(type([2]))
print(type('23'))
```

Output

```
10
4
1
[1, 2, 3, 4]
[4, 3, 2, 1]
23.0
['2', '3']
23
23
<type 'int'>
<type 'list'>
```

```
<type 'str'>
```

7.3 Summary

- There are around 80 builtin functions, complete details is avaiable here https://docs.python.org/2/library/functions.html (https://docs.python.org/2/library/functions.html) .

8.1 Files

- Files are lowest abstraction layer to store data.
- File content can be `text`, `binary` etc ...

8.2 Write

Write a few movie names to a file.

Code

```
filename = 'movies.txt'
f = open(filename, 'w')
f.write('Interstellar\n')
f.write('Inception\n')
f.close()
```

8.3 Read

Read the movie names from the file.

Code

```
filename = 'movies.txt'
f = open(filename)
f.read()
f.close()
```

Output

```
'Interstellar\nInception\n'
```

Read one line at a time from the file.

Code

```
filename = 'movies.txt'
f = open(filename)
for line in f:
    print line
f.close()
```

Output

```
Interstellar

Inception
```

8.4 Context Manager

- Everytime we open a file, we need to close it.
- Python can do this automatically.

Code

```
filename = 'movies.txt'
with open(filename) as f:
    for line in f:
        print line
```

Output

```
Interstellar
Inception
```

8.5 Summary

9.1 Dictionary

- `Dictionary` is `Hash Map or Associative Array` in other programming languages.
- Every entry in dictionary consist of `Key, Value` pair.

9.2 Examples

Code

```
months = {'jan': 31, 'feb': 28, 'mar': 31, 'apr': 30} # Colon is used to separate k
ey and value

# `jan` is key and `31` is value
""" Visual representation of Dictionary
-----------
|Key|Value|
-----------
|jan|31   |
-----------
|feb|28   |
-----------
|mar|31   |
-----------
|apr|30   |
-----------
"""
print(months) # Dictionary don't maintain the order of insertion.
print(months['jan']) # Values in dictionary are accessed using key, in list index
is used.
print(months.get('jan')) # .get returns None if the key is missing
print('dec' in months) # `in` is used to check presence of key in dictionary.
print(len(months)) # len function is used to find total key, value pair in diction
ary.
for key, value in months.items(): # .items() returns two values during every itera
tion. First is key, second is value
    print(key, '->', value)
months['feb'] = 29 # Leap year! if key is already present value will be replaced el
se key, value pair will be added.
print(months)
months['dec'] = 31
print(months)
```

Output

```
{'jan': 31, 'apr': 30, 'mar': 31, 'feb': 28}
31
31
False
4
('jan', '->', 31)
('apr', '->', 30)
('mar', '->', 31)
('feb', '->', 28)
{'jan': 31, 'apr': 30, 'mar': 31, 'feb': 29}
{'jan': 31, 'apr': 30, 'dec': 31, 'mar': 31, 'feb': 29}
```

9.3 Use case

- `Dictionary` is used when representation is key, value pair like `monthly sales, student marks stored subject wise`.
- `Dictionary` lookup takes constant time.

9.4 Summary

- `Dictionary` doesn't maintain the key, value pair order.
- `.get()` method is used to value of the key in dictionary. If `key` is missing `None` is returned.

10.1 Exceptions

- When something goes unexpected interpreter raises `Exception` like dividing by zero, accessing the missing array index.
- `Exceptions` can be caught and acted accordingly.

10.2 try

Code

```
l = [1, 2, 3]
print(l[5]) # This raises IndexError since list only contains 3 elements.
```

Output

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-56-5e770730b19a> in <module>()
1 l = [1, 2, 3]
----> 2 print(l[5]) # This raises IndexError since list only contains 3 elements.


IndexError: list index out of range
```

Code

```
d = {'a': 1}
print(d['b']) # Raises KeyError
```

Output

```
---------------------------------------------------------------------------
    KeyError                                  Traceback (most recent call last)

    <ipython-input-57-945bd7e85bd3> in <module>()
          1 d = {'a': 1}
    ----> 2 print(d['b']) # Raises KeyError


    KeyError: 'b'
```

10.3 catch

Code

```
try:
    l = [1, 2, 3]
    print(l[5])
except IndexError as e:
    print(e)
```

Output

```
list index out of range
```

Code

```
try:
    d = {'a': 1}
    print(d['b'])
except KeyError as e:
    print(e)
```

Output

```
'b'
```

10.4 Catch all

Code

```
try:
    l = [1, 3, 3]
    d = {'a': 1}
    print(l[5], d['a'])
except (IndexError, KeyError) as e:
    print(e)
finally:
    print("end")
```

Output

```
list index out of range
end
```

10.4 Summary

- `try`, `except`, `finally` are three exception handling blocks.
- `except` can catch more than one `Exception` type.

11.1 More Resources

- Learn Python HardWay - http://learnpythonthehardway.org/ (http://learnpythonthehardway.org/)
- Build your search engine - https://www.udacity.com/course/cs101 (https://www.udacity.com/course/cs101)
- Official Docs: https://docs.python.org/2/ (https://docs.python.org/2/)

11.2 Contact

- Email: me@kracekumar.com
- Twitter: https://twitter.com/kracetheking (https://twitter.com/kracetheking)

11.3 Contributors

- Thanks Praseetha (https://twitter.com/void_imagineer) , Mudassir Ali (https://www.facebook.com/lime.4951?fref=ts) for feedback.