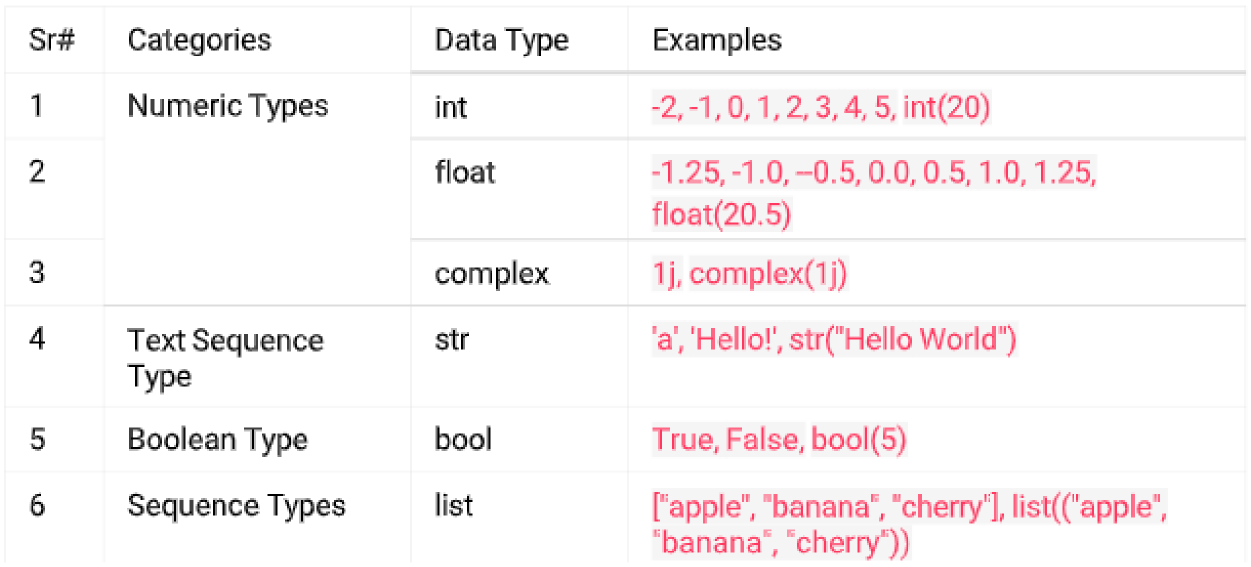
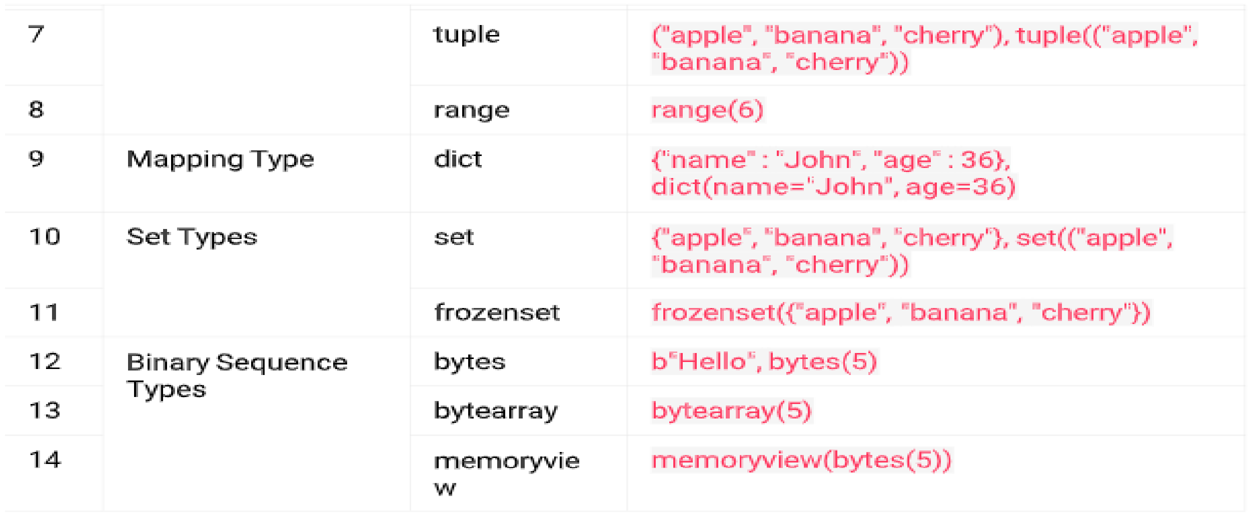
Data Types

The following section describes the standard types that are built into the Python interpreter. These datatypes are divided into different categories like numeric, sequences, mapping etc. Typecasting is also discussed below.

- Bult-in Types

The following chart summarizes the standard data types that are built into the Python interpreter.

Sr#	Categories	Data Type	Examples
1	Numeric Types	int	-2, -1, 0, 1, 2, 3, 4, 5, int(20)
2		float	-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25, float(20.5)
3		complex	1j, complex(1j)
4	Text Sequence Type	str	'a', 'Hello!', str("Hello World")
5	Boolean Type	bool	True, False, bool(5)
6	Sequence Types	list	["apple", "banana", "cherry"], list(("apple", "banana", "cherry"))
7		tuple	("apple", "banana", "cherry"), tuple(("apple", "banana", "cherry"))
8		range	range(6)
9	Mapping Type	dict	{"name" : "John", "age" : 36}, dict(name="John", age=36)
10	Set Types	set	{"apple", "banana", "cherry"}, set(("apple", "banana", "cherry"))
11		frozenset	frozenset({"apple", "banana", "cherry"})
12	Binary Sequence Types	bytes	b"Hello", bytes(5)
13		bytearray	bytearray(5)
14		memoryview	memoryview(bytes(5))

Python has no command for declaring a variable for any datatype. A variable is created the moment you first assign a value to it. Variable names are case-sensitive. Just like in other languages, Python allows you to assign values to multiple variables in one line.

Python Tuples

Tuples are almost exactly the same as lists. They differ in just two ways.

1) The syntax for creating them uses parentheses instead of square brackets.

2) They cannot be modified (they are *immutable*).

Tuples are often used for functions that have multiple return values.

```
t = (1, 2, 3)

t = 1, 2, 3 # equivalent to above

t[0] = 100 # TypeError: 'tuple' object does not support item assignment


# Classic Python Swapping Trick a = 1 b = 0

a, b = b, a # 0 1
```

- Tuple Functions

There are only two tuple methods count() and index() that a tuple object can call.

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5) x = thistuple.count(5)   # 2


thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5) x = thistuple.index(8)   # 3
```

Python Dictionaries

Dictionaries and lists share the following characteristics:

- Both are mutable.
- Both are dynamic. They can grow and shrink as needed.
- Both can be nested. A list can contain another list. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.

Dictionaries differ from lists primarily in how elements are accessed:

- List elements are accessed by their position in the list, via indexing. • Dictionary elements are accessed via keys not by numerical index.

Duplicate keys are not allowed. A dictionary key must be of a type that is immutable. E.g. a key cannot be a list or a dict.

Here are a few examples to create dictionaries:

```
MLB_team = {
    'Colorado' : 'Rockies',
    'Boston'   : 'Red Sox',
    'Minnesota': 'Twins',
    'Milwaukee': 'Brewers',
    'Seattle'  : 'Mariners'
}
# Can also be defined as:
MLB_team = dict([
    ('Colorado', 'Rockies'),
    ('Boston', 'Red Sox'),
    ('Minnesota', 'Twins'),
    ('Milwaukee', 'Brewers'),
    ('Seattle', 'Mariners')
])
# Another way
tel = dict(sape=4139, guido=4127, jack=4098)
```

```
# dict comprehensions can be used to create dictionaries from arbitrar y key and value
expression

{x: x**2 for x in (2, 4, 6)}        # {2: 4, 4: 16, 6: 36}

# Building a dictionary incrementally – if you don’t know all the key-

value pairs in advance person = {}

person['fname'] = 'Joe' person['lname'] = 'Fonebone' person['age'] = 51 person['spouse'] =
'Edna'

person['children'] = ['Ralph', 'Betty', 'Joey']

person['pets'] = {'dog': 'Fido', 'cat': 'Sox'}

# {'fname': 'Joe', 'lname': 'Fonebone', 'age': 51, 'spouse': 'Edna', 'children': ['Ralph', 'Betty',
'Joey'], 'pets': {'dog': 'Fido', 'cat': 'Sox'}}
```

- **Dictionary Modification Examples**

A few examples to access the dictionary elements, add new key value pairs, or update previous value:

```
# Retrieve a value

MLB_team['Minnesota']        # 'Twins'

# Add a new entry

MLB_team['Kansas City'] = 'Royals'

# Update an entry

MLB_team['Seattle'] = 'Seahawks'
```

- **Dictionary Formatting Example**

The % operator works conveniently to substitute values from a dict into a string by name:

```
hash = {}

hash['word'] = 'garfield' hash['count'] = 42

s = 'I want %(count)d copies of %(word)s' % hash  # %d for int, %s for string

# 'I want 42 copies of garfield'
```

- **Dictionary Functions**

The following is an overview of methods that apply to dictionaries:

```
# Let’s use this dict for to demonstrate dictionary functions d = {'a': 10, 'b': 20, 'c': 30}


# Clears a dictionary. d.clear()            # {}


# Returns the value for a key if it exists in the dictionary.

print(d.get('b')) # 20


# Removes a key from a dictionary, if it is present, and returns its v alue.

d.pop('b') # 20


# Returns a list of key-value pairs in a dictionary.

list(d.items())   # [('a', 10), ('b', 20), ('c', 30)] list(d.items())[1][0]       # 'b'
list(d.items())[1][1]     # 20


# Returns a list of keys in a dictionary. list(d.keys())   # ['a', 'b', 'c']


# Returns a list of values in a dictionary.

list(d.values()) # [10, 20, 30]


# Removes the last key-value pair from a dictionary. d.popitem() # ('c', 30)


# Merges a dictionary with another dictionary or with an iterable of key-value pairs.

d2 = {'b': 200, 'd': 400}

d.update(d2) # {'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

For more details, visit iterate dictionary & dictionary comprehensions

Python Sets:

Sets have following characteristics:

- Set in Python is a data structure equivalent to sets in mathematics.
- Sets are a mutable collection of distinct (unique) immutable values that are unordered.
- Any immutable data type can be an element of a set: a number, a string, a tuple.
- Mutable (changeable) data types cannot be elements of the set.
- In particular, list cannot be an element of a set (but tuple can), and another set cannot be an element of a set.
- You can perform standard operations on sets (union, intersection, difference).

Set Initialization Examples

```
You can initialize a set in the following ways:

# Initialize empty set emptySet= set()

# Pass a list to set() to initialize it

dataScientist = set([‘Python’, ‘R’, ‘SQL’, ‘Git’, ‘Tableau’, ‘SAS’])
```

```
dataEngineer = set([‘Python’, ‘Java’, ‘Scala’, ‘Git’, ‘SQL’, ‘Hadoop’])

# Direct initialization using curly braces

dataScientist = {‘Python’, ‘R’, ‘SQL’, ‘Git’, ‘Tableau’, ‘SAS’}

dataEngineer = {‘Python’, ‘Java’, ‘Scala’, ‘Git’, ‘SQL’, ‘Hadoop’}

# Curly braces can only be used to initialize a set containing values

emptyDict= {}
```

Python Lists:

Everything is Python is treated as an object. Lists in Python represent ordered sequences of values. Lists are "mutable", meaning they can be modified "in place". You can access individual list elements with square brackets. Python uses *zero-based* indexing, so the first element has index 0.

Here are a few examples of how to create lists:

```
# List of integers primes = [2, 3, 5, 7]


# We can put other types of things in lists

planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', '

Uranus', 'Neptune']


# We can even make a list of lists hands = [

    ['J', 'Q', 'K'],

    ['2', '2', '2'],

    ['6', 'A', 'K'], # (Comma after the last element is optional)

]

# A list can contain a mix of different types of variables: my_favourite_things = [32, 'AI Lab,
100.25]
```

Good to Know!

Lambda Functions:

Also known as anonymous functions, are concise one-line functions that don't require a `def` statement. They are useful for writing small, single-use functions. Let's see an example of a lambda function:

```
x=lambda n:2*n

print (x (7))
```

Otherwise I would create a function:

```
def dbl(n):

  return 2*n

print(dbl(7))
```

Questions for Practice:

- **Strings from both ends**

Create a string made of the first three and the last three characters

of the original string s and return the new string, so "intelligence" creates "intcne".

However, if the string length is less than 3, return instead the empty string.

- **String jumble**

Create a new string with the help of two input strings a and b. The new string should be separated by a space b/w a and b. Also swap the first 2 chars of each string and return it. e.g.

"mix", "pod" -> "pox mid"

"dog", "dinner" -> "dig donner"