Python Programming

Why Programming?

Python?

Why Learn Programming?

- Improve problem-solving skills
- Building real-life applications
- Better understanding for technologies
- It has great earning potential (Highest salaries) / Strong demands
- Much fun for a lot of us
- Participating in programming competitions (e.g. ICPC/IOI)
  - Build a lot of connections / travel
- You may work in giant companies such as Google and Microsoft

Why Python?

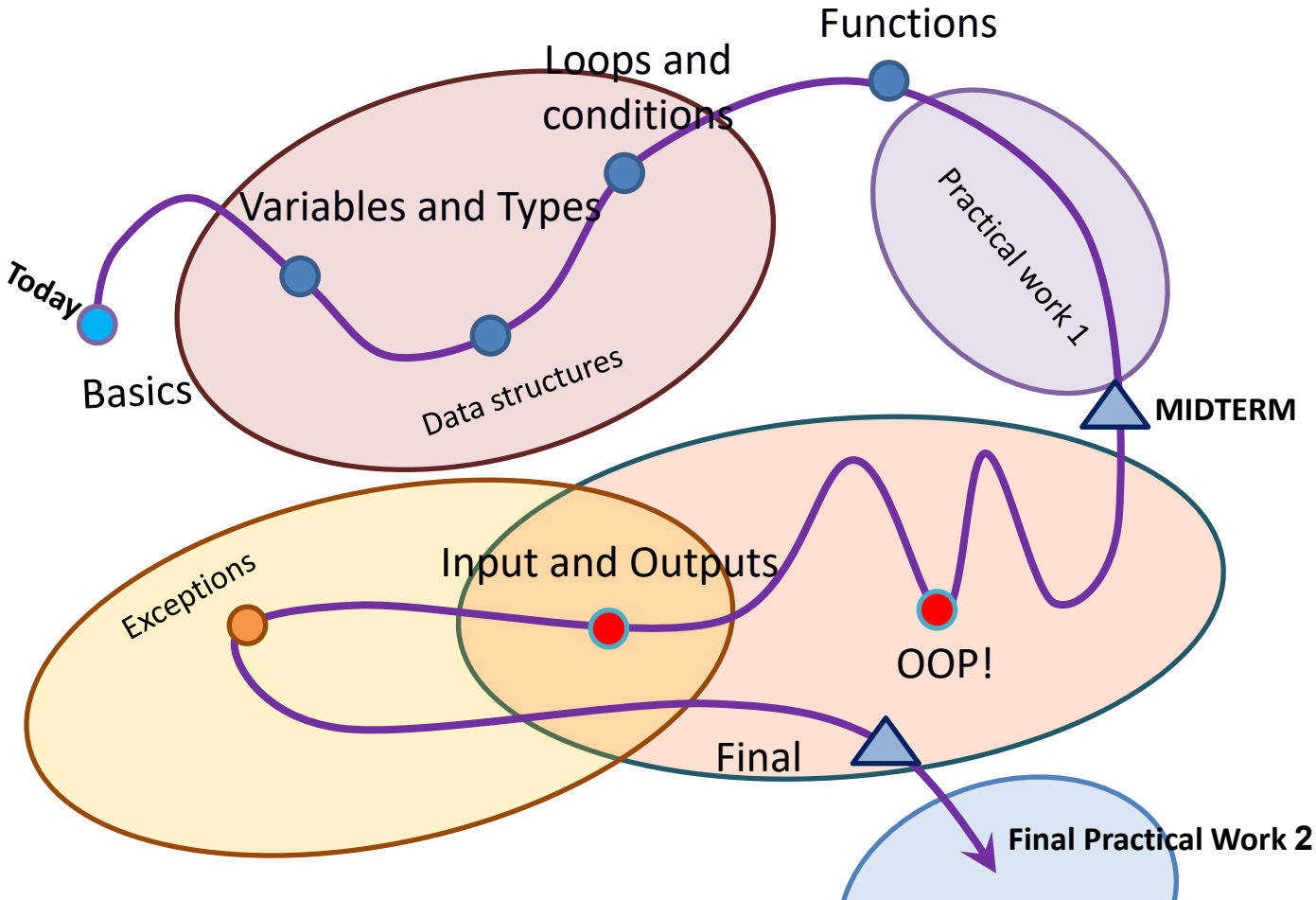
- Powerful general-purpose programming language. Useful in:
  - Data Science, Machine Learning, Web Development, Testing, Automation, Academia, etc
- Simplicity: Easy to install and code! Much easier start for beginners
- Huge Community , open source
  - A lot of questions and answers in the web / stack overflow
  - Python has numerous libraries for different needs
- Python is ranked somewhere in top 3
  - Google: tiobe index, PYPL index, IEEE spectrum programming languages, More
  - Other popular languages: Java, Javascript, C/C++, C#

Math? Who could learn? How hard?

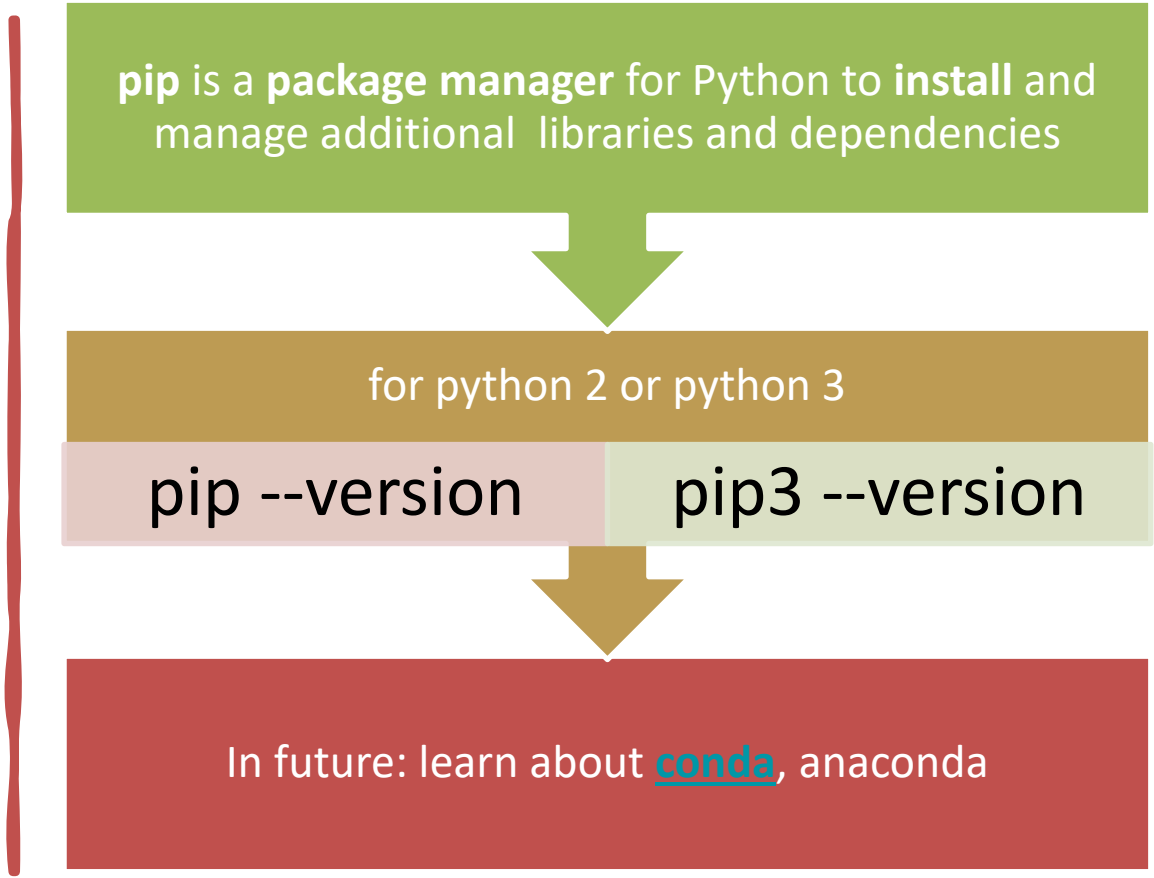
- Programming doesn’t need math background/skills, although useful for mind
- ANYONE can learn programmings (Kids do)
- Learning programming is like learning a new different human language
  - Say you are learning chinese
    - It is so different from Arabic/English/Germany
  - For some students, the begin is a bit weird, then you fall in love.
  - Don’t run away! Don’t tell yourself it is not for me!

Img Src Src Src

Roadmap for course

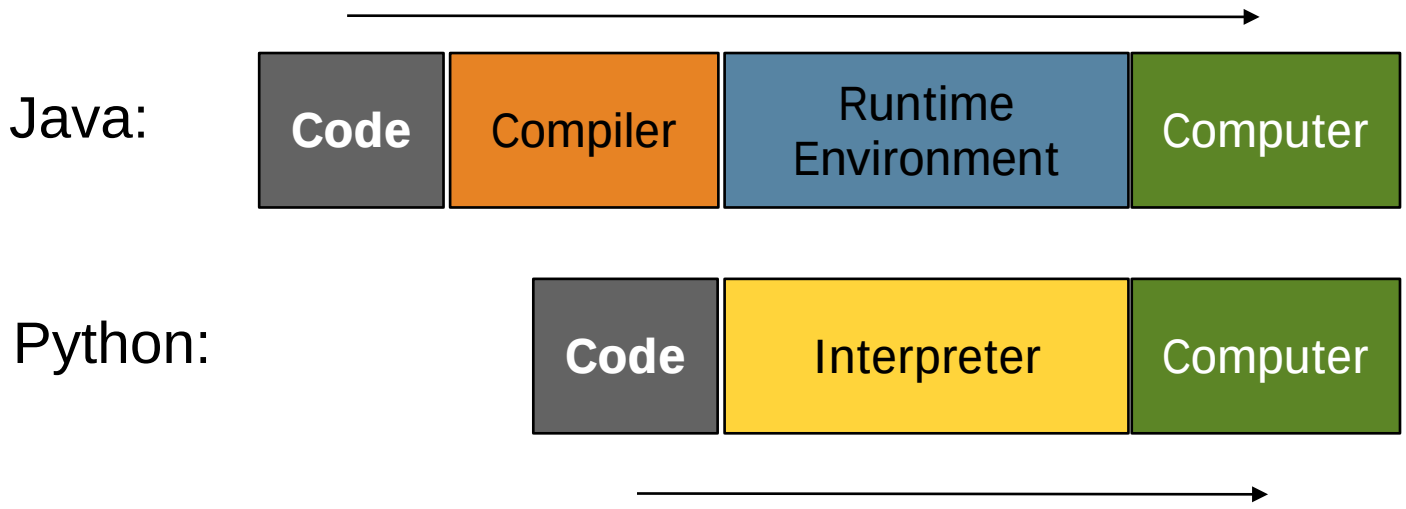


pip


pip is a **package manager** for Python to **install** and manage additional libraries and dependencies
for python 2 or python 3
pip --version
pip3 --version
In future: learn about conda, anaconda

Interpreted Languages

- **interpreted**
  - Not compiled like Java
  - Code is written and then directly executed by an **interpreter**
  - Type commands into interpreter and see immediate results



What's Python ?

- Python is a widely-used, interpreted, object-oriented, and high-level programming language with dynamic semantics, used for general-purpose programming.

Printing

```
# This is comment. Interpreter IGNORE
print('Hello World')            # Hello World
print("Hello World")            # Hello World

# notes
# print is called a function (later)
# ( )       are called parentheses
# '   '     are called single quotes
# "   "     are called double quotes
# H e l W   are called letters :)
# 'Hello World' we call it a string (sequence of letters)
# Hello World   The letter between Hello and World is SPACE
# Above program outputs 2 LINES
```

Behind The scene

- Let’s provide a simple and **informal** to what happens for this line:
  - print(“hello world”)
- The Python interpreter parses your code line by line
- It read a word: print
  - This is a function (command for now) to print something
  - Then it expects after that (“
  - Then it searches till it finds the first “, which should (actually may) have after it)
- It then prints what is between “something”

Printing

```
3   print('Hello')
4   print('World')
5   print('I am Mostafa')
6
7   """
8   triple quotes Allows multiline comments
9   Nothing here is processed by Interpreter
10
11  Output from lines 3-5
12
13  Hello
14  World
15  I am Mostafa
16  """
```

Printing

```
2   print('Hello \nWorld\nI am Mostafa')
3
4   """
5   Hello
6   World
7   I am Mostafa
8
9   \n Things starting with \ are called escape characters
10  \n means add a NEW line now
11  """
12
13  print(' Hello \n  World\n   I am Mostafa')
14  """
15   Hello
16    World
17      I am Mostafa
18  """
```

Printing

```
3   print('Hello '
4         'World '
5         'I am Mostafa')
6
7   """
8   Hello World I am Mostafa
9
10  Line 3-5 are a SINGLE command for print
11      We can split on several lines
12  """
13
```

Doing Arithmetics

```
2   # we can do simple arithmatic!
3   print(1+2+3)
4   print(3 * 4)
5   print      (6 / 2)
6   print(7 / 2)
7   print('7 / 2')      # this is a normal print msg
8
9   """
10  6
11  12
12  3.0
13  3.5
14  7/ 2
15  """
```

```
3
4   print("There is", 1, "instructor not", 2 * 3)
5
6   # There is 1 instructor not 6
```

Comma in Printing

- , is called comma
- Comma prints space between different items

Data type: We vs Programming

Value	We call it	Data Type
52 *(I am 52 years old)*	Number	Integer
12**.7** *(The baby is 12.7 kg)*	Number	Floating point
Computer	Word	String / Sequence of characters
Male of Female?	Status of 2 things	Boolean (2)

Data types: integer, floating point, string

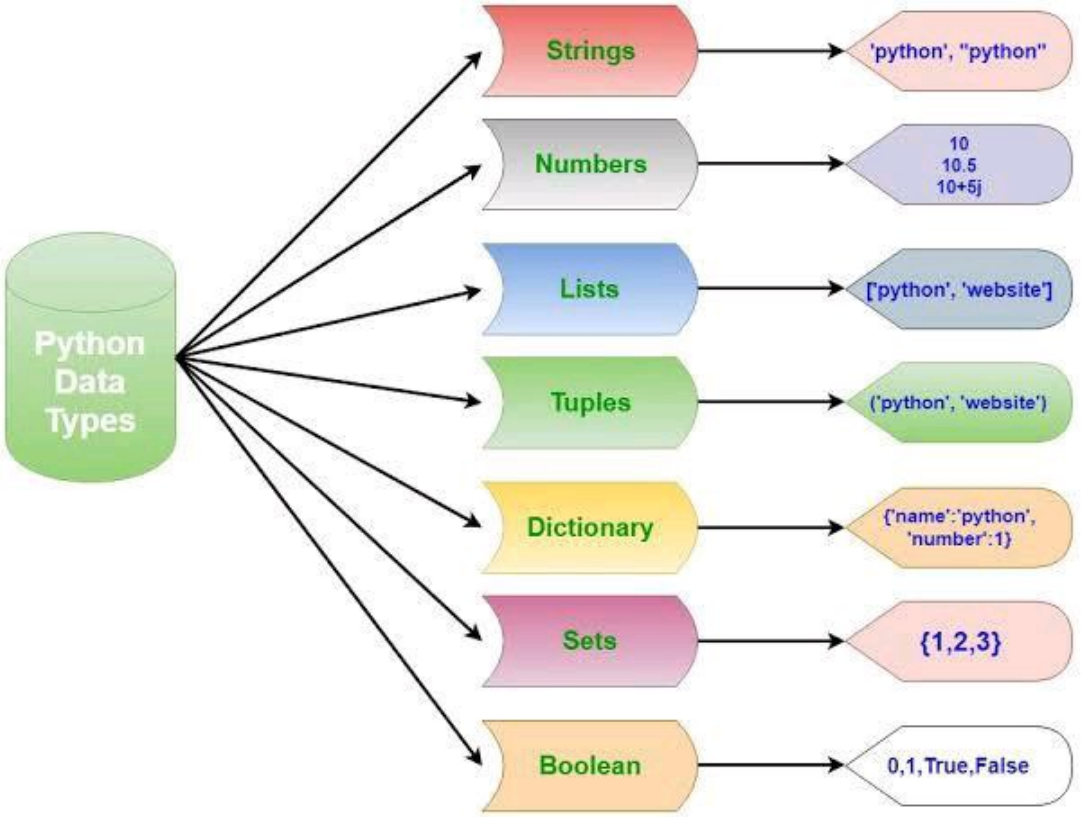
- integer
  - 10, 200, 100, 200, 1000000
- Floating point (has .)
  - 10.5, 200.0, 2000000.1
- String
  - ‘Hello’, “World”, ‘2000’, “3000”, ‘اهلا بكم’

Data type: Boolean

- Some things are of two types only. E.g. a person is Male of Female?
  - Programmers call them **boolean**. Their values are true or false
  - E.g. Male is True, Femail is False (or the opposite)
  - E.g. coin is head or tail. We can think head is true and tail is false
  - E.g. Light can be on or off. Computers "understand" on and off.
    - On = True
    - Off = False

More data types later

Img src



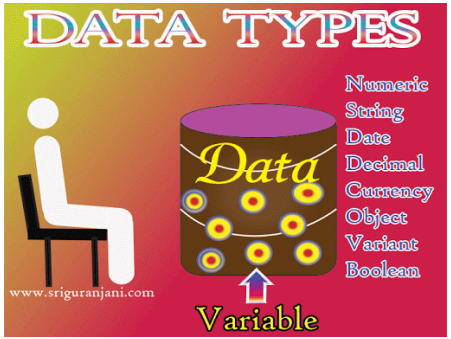
The need for names!

- Let say we want to build a program for hospital
  - We need to refer to patients
  - One of them is called “Mostafa”. He is 55 years old. He has 2 children!
  - We also has a Doctor: who has name, salary, address, etc
- How can we represent this information?
  - We need to put them in memory when the program starts
  - We need to have names to **refer** to them?
  - E.g. I want to know mostafa’s age?

Computer Memory Like Streets

- Each home has street address (location)
  - Name and number
  - 37 John Street
  - 37 = location
  - John = Name
- There are people in the home
  - People have types
    - Male, Femal, Child

Img src



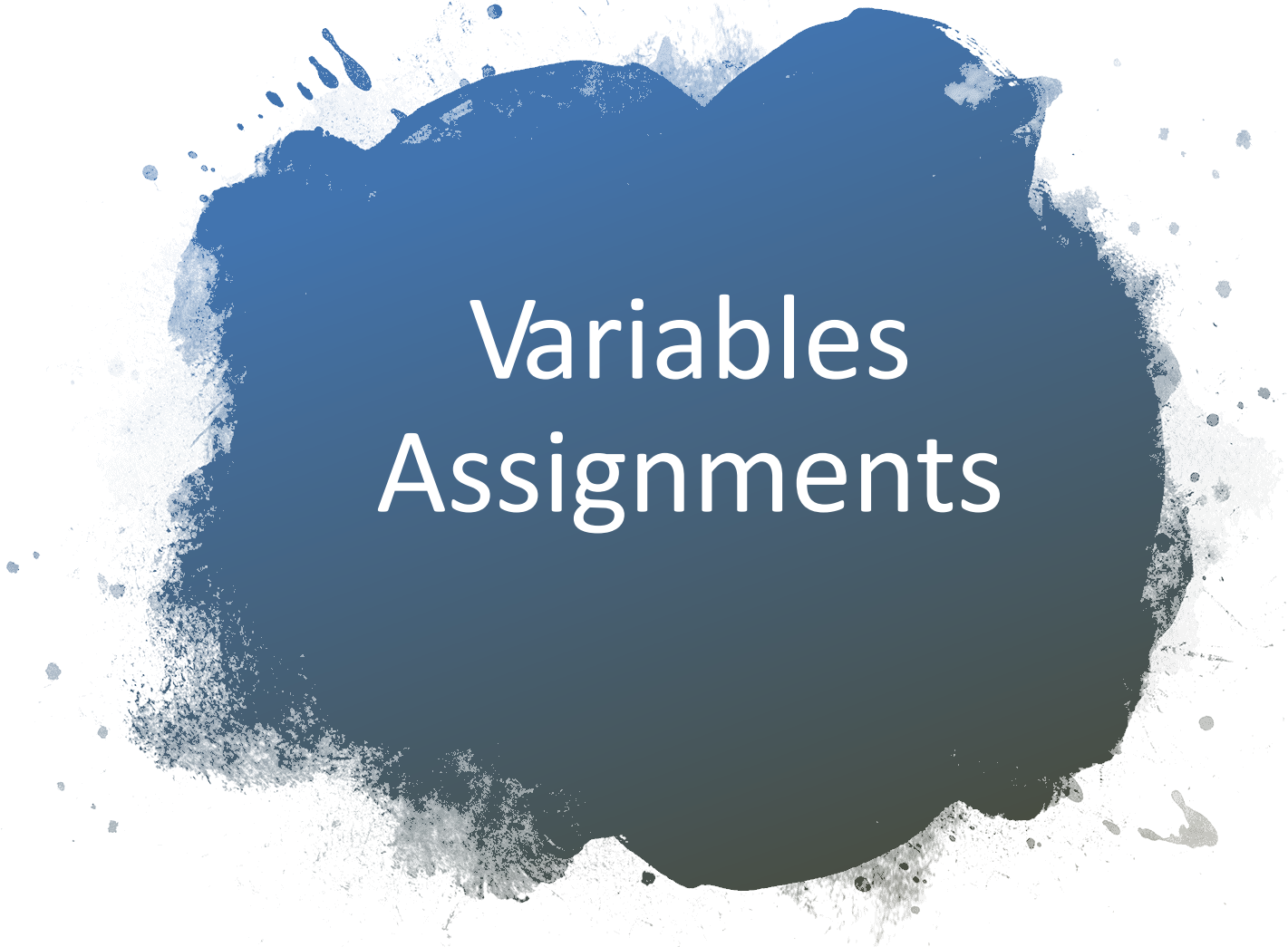
Computer Memory (RAM)

- Think of it as boxes
- Each box has
  - Address location
  - Type (e.g. integer or floating point)
  - Name: E.g. age, name or salary
- We call it **variable**
  - A box in the memory

1	EMPTY	NA
2	**Age = 55**	Integer
3	**Weight = 92.5**	Floating point
4	**Gender = Male**	Boolean
5	**Name = “Mostafa”**	String

So

- Age = 52
- Variable
  - **Name** (aka identifier) = Age
  - **Value** = 52
  - **Datatype** = Integer
  - **Memory address** = Somewhere in RAM


Variables
Assignments

Variables Assignments

- We will learn how to create, assign and change the value of variables!
- *Take a minute to read the program*
    - Maybe make a guess :)
- Initially, the memory has nothing
- Let's run the interpreter
- It goes **line by line** to execute it
- Initially, we have an empty RAM

```
3   print(55)       # 55
4
5   # Create variable
6       # in some memory location
7   # Its name it age (identifier)
8   # Its current type is integer
9   age = 55
10      # = means assign a value 55
11
12  print(age)      # 55
13  print(age + 5)  # 60
14
15  # A floating variable named weight
16  # Assign value 75.8
17  weight = 75.8
18
19  print(weight)   # 75.8
```

Variables Assignments

- **Line 3:**
  - Just print 55
  - No effect on RAM

```
3  print(55)        # 55
4
```

Memory Before

Memory After

Variables Assignments

- **Line 9:** age = 55
  - Create variable in the memory
    - Name = age
    - Value = 55
    - Type = int
- When you see var **=** something
  - = means assign right value to left var
- Each variable has a name
  - We call it identifier
  - Has some memory location (0x2045)

```
5   # Create variable
6       # in some memory location
7   # Its name it age (identifier)
8   # Its current type is integer
9   age = 55
10      # = means assign a value 55
```

Memory Before

Memory After

age: 55 (type int)

Variables Assignments

- **Line 12:** print(age)
  - Print receives age, which is var
  - Its value in memory 55
  - So equal to
  - print(55) ⇒ 55
- **Line 13:** print(age + 5)
  - Print receives **age + 5**
  - **age** + 5 is an expression
  - What is **age** in memory? 55
  - So what us **age**+5? 55 + 5
  - Print(60) ⇒ 60
- No memory effects

```
12    print(age)      # 55
13    print(age + 5)  # 60
```

Memory Before

age: 55 (type int)

Memory After

age: 55 (type int)

Variables Assignments

- **Line 17:** weight = 75.8
  - Create another variable in the memory
    - Name = weight
    - Value = 75.8
    - Type = float
- **Line 19:** print(weight)
  - print(75.8) ⇒ 75.8

```
15  # A floating variable named weight
16  # Assign value 75.8
17  weight = 75.8
18
19  print(weight)   # 75.8
```

Memory Before

age: 55 (type int)

Memory After

age: 55 (type int)
weight: 75.8 (type float)

Limits

- What is the maximum number of characters in a string?
  - No limit. Up to your machine physical memory
- What about int?
  - *In python 3: there is **no limit** on the maximum value to use an integer*
  - *193485903485904385903459834098590348508439509348 98…..43985798435*
- What about float?
  - Up to **1.79**76931348623157 * 10^{308}
    - Approximately 2 followed by 308 zeros = That is a 309 digits number
  - Consider that: float is an approximated number. Not accurate!
  - **import** sys
  - print(sys.float_info.min, sys.float_info.max)
    - 2.2250738585072014e-308 1.7976931348623157e+308


Identifier

Identifier (variable name)

- Identifier: Variable name: sum = 10 => sum is identifier
- Identifier consist of: letters, digits, _
  - status1, total_sum, _valid, is_valid
- It can't start with digit: [7Core is wrong]
- It can't contain space or **: , < > / \ ? ! ( ) @ # $%^&~+-*** [my-var is wrong]
- It is case sensitive: sum != SUM
- You shouldn't use reserved keyword (built in)
  - **and**, **as**, **assert**, **break**, **class**, **continue**, **def**, **del**, **elif**, **else**, **except**, **finally**, **for**, **from**, **global**, **if**, **import**, **in**, **is**, **lambda**, **not**, **or**, **pass**, **raise**, **return**, **try**, **while**, **with**, **yield**
- Common guideline: Use a lowercase single letter, word, or words.
  - Separate words with underscores to improve readability. (e.g. total_sum)

Some from software Engineering

- Readability
  - A good code should be easy to read`
  - Think in meaningful variable names
    - s = 0 ⇒ bad name: says nothing
    - I may use such names for shorter coding or educational purposes. You don't
    - total_sum = 0 ⇒ good name: seems something will be summed (e.g. salaries)
- Code Review
  - When you finish a coding task in a company. One or more guys review it
  - Code needs to be correct + readable + following coding guidelines
- Coding guidelines
  - A list of dos and don'ts for coding programs.
  - Google Python Style Guide


Dynamic Typing

Variables Assignments

- In python, NOT only we can change the value, we can even change the type!
  - It is called Dynamic Typing
- *Take a minute to read the program*
  - Maybe make a guess :)
- Let's trace it

```
3   var = 55
4
5   # change value in memory
6   var = 10
7   print(var)          # 10
8
9   # Get, Change, Assign
10  var = var + 5
11  print(var)          # 15
12
13  # Change the DATA type to floating po
14  # Python uses Dynamic typing: Change
15  # Set memory value to 25.5
16  var = 6.5           # 6.5
17  print(var)
18
19  var = 3 * var - var
20  print(var)          # 13.0
21
22  # Change type again to string
23  var = 'Dr Mostafa'
24  print(var)          # Dr Mostafa
```

Variables Assignments

```
3    var = 55
4
```

- **Line 3:**
  - Create variable
    - Name = var
    - Type = int
    - Value = 55

Memory Before

Memory After

Var: 55 (type int)

Variables Assignments

- **Line 6:**
  - Re-assign value 10 to variable var

```
3   var = 55
4
5   # change value in memory
6   var = 10
7   print(var)          # 10
8
```

Memory Before

Var: 55 (type int)

Memory After

Var: **10** (type int)

Variables Assignments

- **Line 10**
  - var = var + 10
  - Evaluate expression var + 10
  - What is var's value in memory? 10
  - What is var + 5? 10 + 5 = 15
  - Re-assign value 15 to var

```
 9    # Get, Change, Assign
10    var = var + 5
11    print(var)          # 15
12
```

Memory Before

Var: 10 (type int)

Memory After

Var: **15** (type int)

Variables Assignments

- **Line 16:**
  - Re-assign value 6.5 to variable var
    - But **change type** to float

```
13  # Change the DATA type to floating point
14  # Python uses Dynamic typing: Change types
15  # Set memory value to 25.5
16  var = 6.5           # 6.5
17  print(var)
```

Memory Before

Var: 15 (type int)

Memory After

Var: **6.5** (type float)

Variables Assignments

- **Line 19**: Re-assign var to expression **3 * var - var**
  - What is var's value in memory? 6.5
  - What is 3 * var - var? 3 * 6.5 - 6.5 = 13.0
  - Re-assign value 15 to var

```
19    var = 3 * var - var
20    print(var)          # 13.0
21
```

Memory Before

Var: **6.5** (type float)

Memory After

Var: **13.0** (type float)

Variables Assignments

- **Line 16**
  - Re-assign value ‘Dr Mostafa’ to variable var
    - But **change type** to string

```
22  # Change type again to string
23  var = 'Dr Mostafa'
24  print(var)          # Dr Mostafa
```

Memory Before

Var: **13.0** (type float)

Memory After

Var: **‘Dr Mostafa’** (type string)

Dynamic Typing

- Recall type: int, float, str, etc
  - Can we **change** the variable's type from a line to another?
  - Does the language **check** that: before or during running the code?
- Python is a **dynamically** typed language.
  - The types of variables are checked **during running the program**,
  - **Pros**: Easy development
  - **Cons**: More **errors** at runtime and in shipped code
  - Future reading
- Other languages are called: **statically** typed language
  - Every variable has a specific type that CAN'T be changed (checked **before running**)
  - Example: C++ and Java

String Manipulation

String Manipulation

- Optional reading about Tab

```
print('I am mostafa')                # I
print('I am' + ' ' + 'mostafa')      # I

str1 = 'I am'
str2 = ' mostafa'
print(str1 + str2)                   # I

print(str1 * 3)                # I amI amI
print(2 * str1 + str2)         # I amI am

str1 = 'Hello '
str3 = str1 + str2 + str1
print(str3)                    # Hello  mo

# \t is escape character for TAB
# A Tab character shifts over to the ne
# By default, there is one every 8 spac
print('Hello\tworld')          # Hello
print('Hello\t\tworld')        # Hello
```

Using “”””

```
"""hello world"""              # comment
print('''hello world''')       # hello world
print("""hello world""")       # hello world

# hello let's learn "a lot" world
print("""hello let's learn "a lot" world""")

"""
Easily print
on several lines

using me
"""
print("""Easily print
on several lines

using me
""")
```

Python Programming

List Data Structure

Write a program that:

- reads 1000 integers and print them reversed!
- reads 1000 integers and find pairs of numbers with sum 12345?
- We can define 1000 variables! But this is a crazy idea!
- Python provides a **list** data structure that allows us to have be objects of many data types in a convenient way
    - **Data structures** are containers that organize and group data types together in different ways.
- List: is an ordered sequence of items
    - They can be of different types!
    - It is a **mutable** class

Creating a list

- We use brackets [] to create a list
- Between []: put your items, comma separated
- List is a sequence of items
  - It is iterable
  - We can use in operator to iterate
- A list can have different data types!

```
2
3   my_list = [1, 2, 3, 4]
4
5   print(len(my_list))      # 4
6
7   print(2 in my_list)      # True
8   print(9 in my_list)      # False
9
10  for item in my_list:     # 1 2 3 4
11      print(item, end=' ')
12  print()
13
14  print(my_list)  # [1, 2, 3, 4]
15
16  # list of different data types!
17  my_list = [1, 'mostafa', 4.5]
18
19  #IndexError: list assignment index out of range
20  #my_list[3] = 0
```

Indexing

```
1
2   numbers = [10, 2, 7, 5, 3]
3
4   numbers[0] = 9
5   numbers[2] *= 3
6   numbers[4] += 1
7
8   print(numbers[4])
9
10
11  for idx in range(5):
12      print(numbers[idx], end=' ')
13      # 9 2 21 5 4
```

- Line 2 creates a list

Index	0	1	2	3	4
numbers	10	2	7	5	3

- Line 4 changes first number to 9

Index	**0**	1	2	3	4
numbers	**9**	2	7	5	3

- Line 5 and 6 also do changes

Index	**0**	1	**2**	3	**4**
numbers	**9**	2	**21**	5	**4**

*+ and * operators*

```
1
2   my_list = [1, 'mostafa', 4]
3
4   another_list = [99, 11.5]
5
6   # 1 mostafa 4 99 11.5
7   conc_list = my_list + another_list
8
9   # 99 11.5 99 11.5
10  thrd_lst = 2 * another_list
11
12  lst = [0] * 6   # 0 0 0 0 0 0
13
14  lst += [2, 3] + [5]
15  # 0 0 0 0 0 0 2 3 5
16
```

Mutability

```
>>> my_lst = [1, 2, 3, 4, 5]
>>> my_lst[0] = 'one'
>>> print(my_lst)
['one', 2, 3, 4, 5]
```

You can replace 1 with 'one' in the above list. This is because lists are **mutable.**

- **Mutability** is about whether or not we can change an object once it has been created. If an object (like a list or string) can be changed (like a list can), then it is called **mutable**. However, if an object cannot be changed with creating a completely new object (like strings), then the object is considered **immutable**.

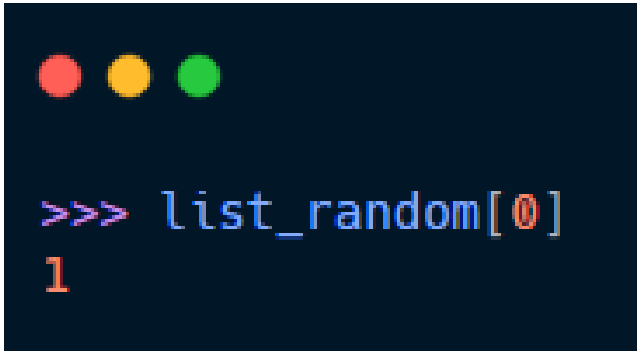
```
greeting = "Hello there"
greeting[0] = 'M'
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: 'str' object does not support item assignment
```

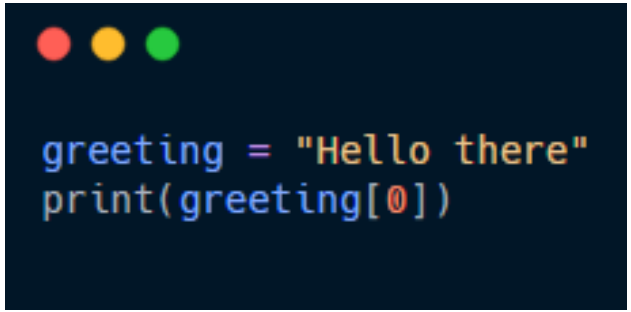
This is because strings are **immutable**. This means to change this string; you will need to create a completely new string.

Order

- **Order** is about whether the position of an element in the object can be used to access the element. **Both strings and lists are ordered.** We can use the order to access parts of a list and string.

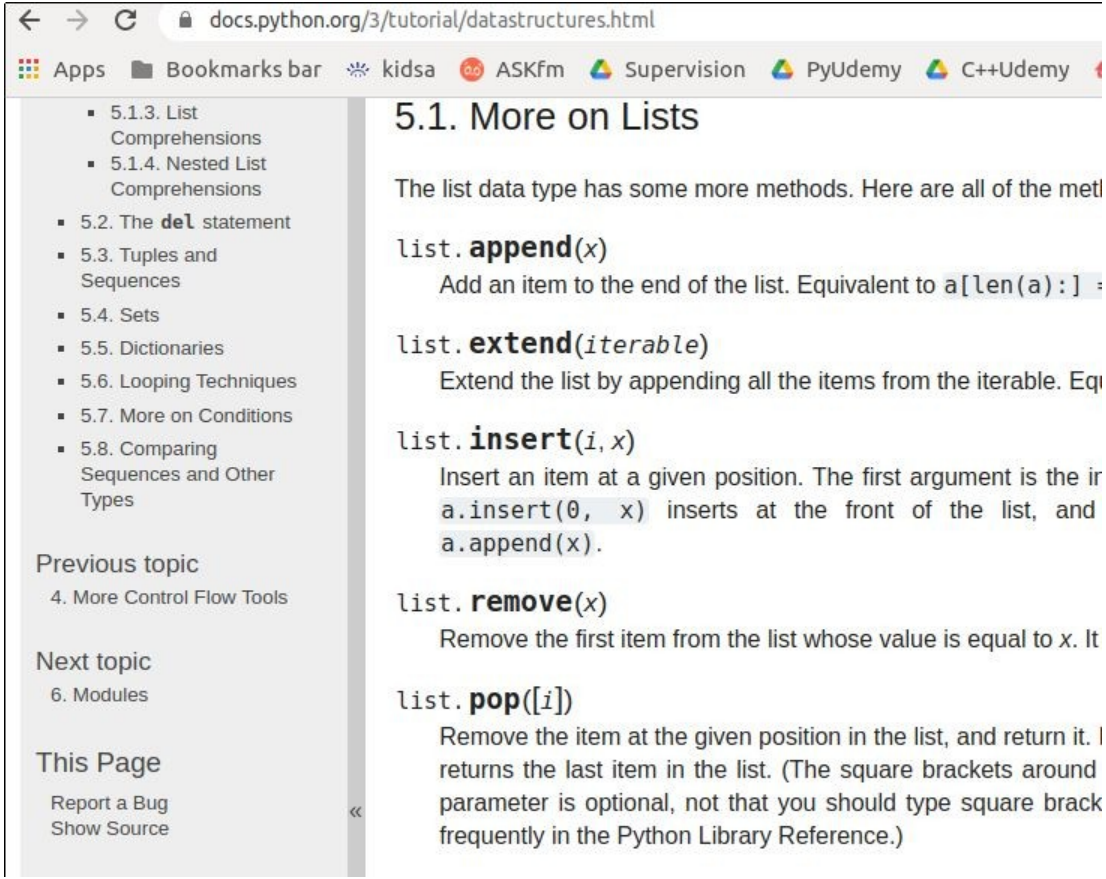
You will see some data types that will be unordered.






List Methods

Documentation



Append, extend and insert

```
2   my_list = [1, 5, 10, 17, 2]
3
4   # append: add item to the end
5   my_list.append('Hii')    # 1 5 10 17 2 Hii
6
7   # Extend the list by appending all the items from the iterable
8   another_lst = [3, 1]
9   my_list.extend(another_lst)
10  # 1 5 10 17 2 Hii 3 1
11
12  #TypeError: 'int' object is not iterable
13  #my_list.extend(2)
14
15  # Insert an item at a given position
16  my_list.insert(2, 'Wow')
17  # 1 5 Wow 10 17 2 Hii 3 1
18
19  for item in my_list:
20      print(item, end=' ')
21  print()
22
```

Pop, remove + del statement

```
my_list = [1, 5, 10, 17, 2, 'Hii']

# pop removes the item at a specific index and returns it.
print(my_list.pop())    # Hii  - default last item
# Now list is : 1 5 10 17 2

print(my_list.pop(3))    # 17
# Now list is : 1 5 10 2

# del removes the item at a specific index:
del my_list[0]  # 5 10 2

# remove removes the first matching value, not a specific index:
my_list.remove(10)  # 5 2

# ValueError: list.remove(x): x not in list
#my_list.remove('Hei')
```

Index and clear methods

```
3   my_list = [1, 15, 7, 'mostafa', 7, True, 0]
4
5   # search and return the FIRST index
6   print(my_list.index(7))         # 2
7   print(my_list.index('mostafa')) # 3
8   print(my_list.index(True))      # 0 **
9   print(my_list.index(False))     # 6 **
10
11  #ValueError: 'Wow' is not in list
12  #print(my_list.index('Wow'))
13
14  my_list.clear()
15  print(len(my_list))     # 0
16
```

Count method

```
2
3   my_list = [4, 5, 7, 4, 5, 4, 8]
4
5   print(min(my_list), max(my_list))   # 4 8
6
7   print(my_list.count(4))             # 3
8   print(my_list.count([4, 5]))        # 0 **
9
10  my_list = ['ali', 'ALI', 'ali']
11
12  print(my_list.count('ali'))         # 'ali'
13
14
```

+ vs +=

- += is changing in-place
- This will imply differences behind the scene
- They only similar in binding
  - UnboundLocalError

```
lst = [1, 2, 3]
print(id(lst))      # 0x111
lst += [4]          # in-place change     - internally: __iadd__
print(id(lst))      # 0x111
lst = lst + [5]     # NEW memory creation - internally: __add__
print(id(lst))      # 0x222

# += is behaving similar to .extend

lst += ['Hey']  # iterate on list: add 1 item
print(lst)      # [1, 2, 3, 4, 5, 'Hey']

lst += 'Hey'    # iterate and add 3 items
print(lst)      # [1, 2, 3, 4, 5, 'Hey', 'H', 'e', 'y']

#TypeError: can only concatenate list (not "str") to list
#lst = lst + 'Hey'
#lst = lst + 10
```

Sorting and Reversing Methods

Sort Method

- Algorithm: is a step-by-step procedure for calculations
  - We already tried several algorithmic (computational) problems in loops section
- Sort algorithm: Order the items. By default from small to large
- **In-place algorithm**: It doesn't create new memory. I modifies the given one
  - Minor memory creation may occur

```
2   lst = [5, 7, 2]
3   # NO new list: in-place - memory efficient
4   lst.sort()                  # [2 5 7]
5
6   lst.sort(reverse=True)      # [7 5 2]
7
8   # common mistake:
9   lst = lst.sort()
10  # lst now is NONE!
11
```

Sorted ***function***

- Returns a sorted list of the specified **iterable** object.
  - Work on list and others

```
12  lst = [5, 7, 2]
13  lst_sorted_cpy = sorted(lst)    # sorted copy
14  # lst = NO CHANGE
15  # lst_sorted_cpy [2 5 7]
16
17  my_str = 'zacb'
18  new_lst = sorted(my_str)    # LIST! ['a', 'b', 'c', 'z']
19  new_lst = sorted(my_str, reverse=True)
20  # new_lst = ['z', 'c', 'b', 'a']
21
22  print(new_lst)
23  # common mistake
24  sorted = sorted(my_str)
25  # now sorted become a variable. You can't call the function
26  # TypeError: 'list' object is not callable
27  #sorted = sorted(my_str)
```

Confusing Trick Questions
Sort vs Sorted

- The primary difference between the list sort() function and the **sorted()** function is that the **sort()** function will modify the list it is called on. The sorted() function will create a new list containing a sorted version of the list it is given.

```
position = ['Software Engineer', 'Hardware Engineer', 'Data Engineer']

new_list = sorted(position)

# new_list = ['Data Engineer', 'Hardware Engineer', 'Software Engineer']
print(new_list)

# position = ['Software Engineer', 'Hardware Engineer', 'Data Engineer']
print(position)

position.sort()

# position = ['Data Engineer', 'Hardware Engineer', 'Software Engineer']
print(position)
```

Reverse Method .... Reversed Function

```
2  my_list = [1, 2, 3, 4]
3
4  my_list.reverse()   # 4 3 2 1
5
6  my_list += ['Hi']
7
8  new_lst = reversed(my_list)
9  print(new_lst)  # list_reverseiterator
10
11 new_lst_rev1 = list(reversed(my_list))
12 print(new_lst_rev1)  # list_reverseiterator
13 # ['Hi', 1, 2, 3, 4]
14
15 new_lst_rev2 = my_list.copy()
16 new_lst_rev2.reverse()
17 print(new_lst_rev2)
18 # ['Hi', 1, 2, 3, 4]
19
```

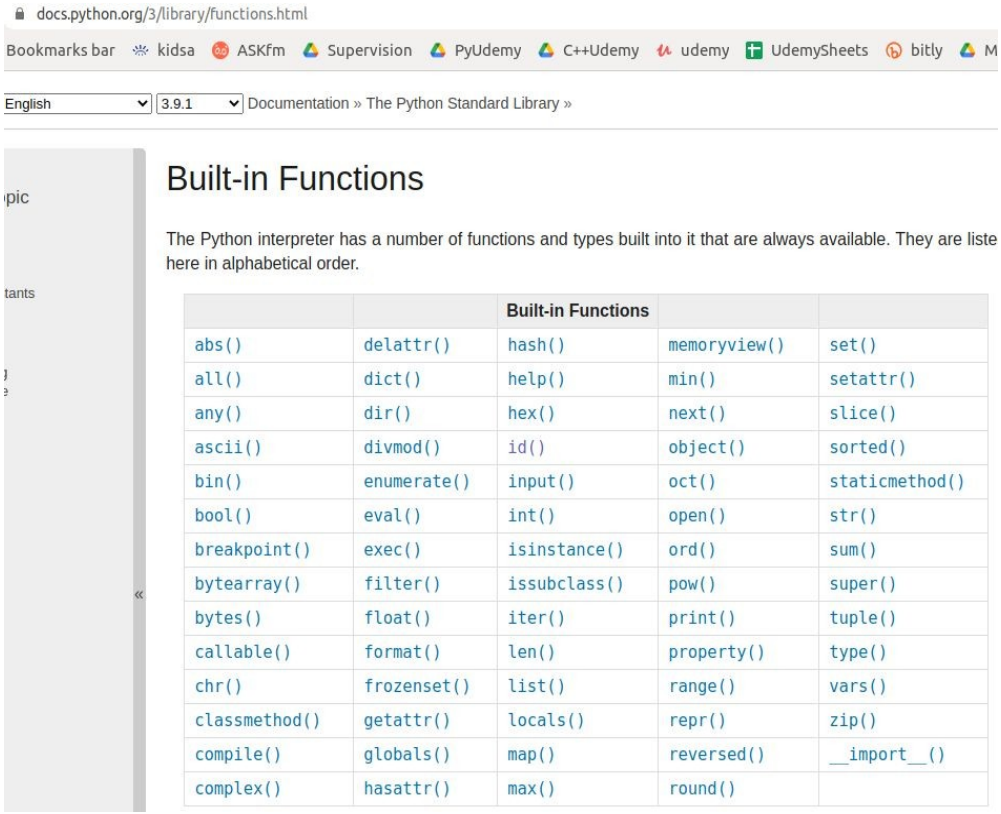
Iterate in a reversed order

```
2   lst = [7, 8, 9, 'Hi']
3
4   for pos in range(len(lst)): # C++/Java style
5       print(lst[len(lst) - pos - 1], end=' ')
6   print()
7
8   # Python: range(start, end, step)
9   for idx in range(len(lst) - 1, -1, -1):
10      print(lst[idx], end=' ')
11  print()
12
13  # Better
14  for item in reversed(lst):  # NO copy is created
15      print(item, end=' ')
16  print()
17
18  for pos, item in reversed(list(enumerate(lst))):
19      print(pos, item, end=' - ')
20      # 3 Hi - 2 9 - 1 8 - 0 7
21
22      # be careful:
23      # list(iterable) => makes copy: more memory / slower
24
```


List with
Functions

Built-in Functions

- We already know: *abs, min, id, enumerate, input, bool, int, str, sum, pow, float, print, len, type, range, globals, map, max, round*
- Some of them have relations with list



		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Flexible Reading

```
2   # .split() return list of strings
3   my_list = input().split()
4
5   for item in my_list:
6       print(item, end=' ')
7   print()
8
9   # now list of integers
10  my_list = list(map(int, input().split()))
11
12  print(type(my_list), type(my_list[0]))  # list, int
13
14  for item in my_list:
15      print(item, end=' ')
16  print()
17
18  # very helpful to read variable number of items on same line
19
```

sum, min, max, help functions

```
2
3   my_list = [4, 5, 7, 4, 5, 4, 8]
4
5   print(sum(my_list))      # 37
6
7   print(min(my_list), max(my_list))    # 3 8
8
9   my_list = ['ali', 'ziad', 'mostafa']
10
11  print(min(my_list), max(my_list))    # ali ziad
12
13  help(my_list.count)      # without () : passing a function
14
15  """
16  Help on built-in function count:
17
18  count(value, /) method of builtins.list instance
19      Return number of occurrences of value.
20  """
```

enumerate function

```
my_list = [1, 'mostafa', 4]
for idx, item in enumerate(my_list):
    print(idx, item)
    idx = -100  # no effect
"""
0 1
1 mostafa
2 4
"""

# NOTE: this creates a complete list in memory
# Slow for a huge range
lst = list(enumerate(range(5, 9)))

for item in lst:
    print(item)
"""
(0, 5)
(1, 6)
(2, 7)
(3, 8)
"""
```

all

```python
# all: Return True if all elements of the iterable are true
lst = [10, 20, -12, 'Mostafa']

print(all(lst))      # True
print(all([]))       # True

# items cause Faulse
print(all([False]))      # False
print(all(['']))         # False
print(all([0]))          # False

print(all([10, 0, 2]))       # False
```

Comparison

```
# same rules as string comparison
    # Item by item comparison
        # if an item is smaller, its list is smaller
        # if one list ended, it is the sammer
lst1 = [1, 5, 8]
lst2 = [1, 5, 8]
lst3 = [1, 5]
lst4 = [7, 5]
print(lst1 is lst2)     # False (must)
print(lst1 == lst2)     # True

print([1, 2, 3] is [1, 2] + [3])    # False (must)
print([1, 2, 3] == [1, 2] + [3])    # True


print(lst1 < lst2)      # False
print(lst1 <= lst2)     # True
print(lst1 <= lst3)     # False
print(lst1 <= lst4)     # True

# TypeError: '<' not supported between instances of 'int' and 'str'
#print([1, 2] < ['mostafa'])
```

Python Programming

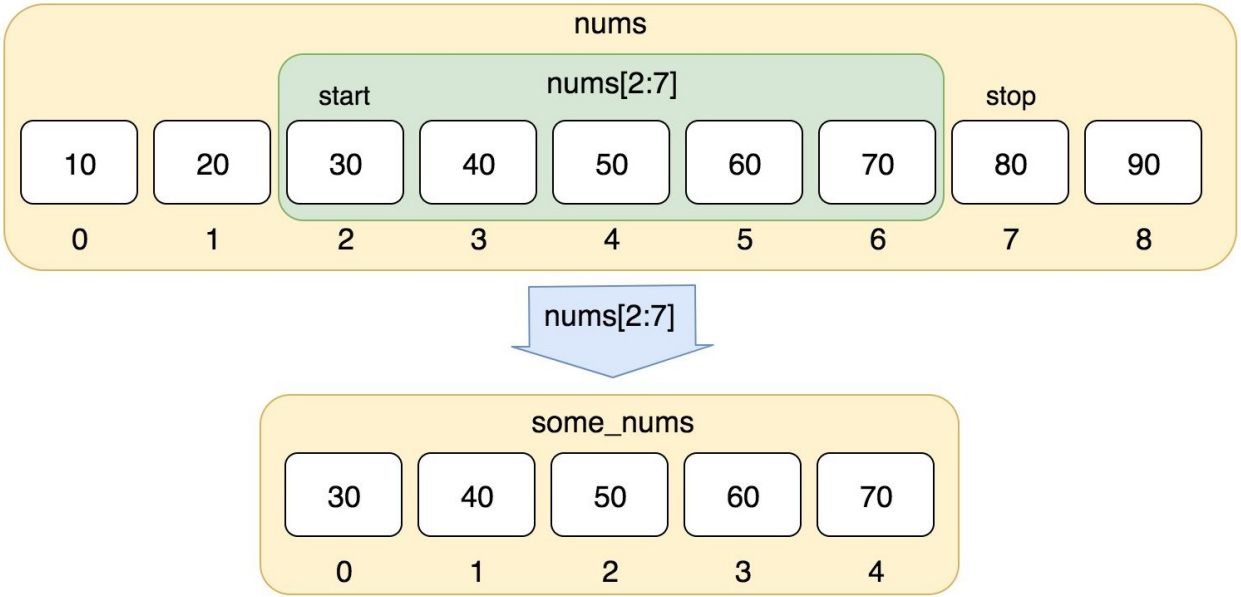
Slicing

Recall: Range Function

```
# [First index to include, first index to exclude : step]
# step can be +ve or -ve
# range can be increasing or decreasing based on step

print(type(range(5)))

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

print(list(range(2, 5)))             # [2, 3, 4]

print(list(range(1, 21, 4)))         # [1, 5, 9, 13, 17]

print(list(range(5, 0, -1)))         # [5, 4, 3, 2, 1]

print(list(range(10, 0, -2)))        # [10, 8, 6, 4, 2]

print(list(range(5-1, -1, -1)))      # [4, 3, 2, 1, 0]
```

Slicing

- Slicing is a flexible tool to build new lists out of an existing list.



Img src

Slicing: <first index to **include**, first index to **exclude**>

```
3   my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
4
5   # 2 is the start
6   # 6 is end (exclusive): ends actually at 5
7   sub_list = my_list[2:6]      # 2 3 4 5
8
9   sub_list[0] = 100            # my_list is NOT changed
10
11  sub_list = my_list[5:6]      # 5 a single element
12
13  sub_list = my_list[5:1000]   # 5 6 7 8
14
15  # syntax: my_list[start : end+1
16
```

Default limits

```
2   my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
3
4   sub_list = my_list[0:5]  # 0 1 2 3 4
5   # If you did not provide start: then 0
6   sub_list = my_list[ :5]  # 0 1 2 3 4
7
8   # 9 = len(my_list)
9   sub_list = my_list[4:9]  # 4 5 6 7 8
10  # similarly: if not end: it is len
11  sub_list = my_list[4: ]  # 4 5 6 7 8
12
13  # observe:
14  # my_list[4] is the 5th element (index 4)
15  # my_list[4:] is slice from index 4 to last element
16  # my_list[:4] is slice from 0 to 3
17
18  same_values = my_list[:4] + my_list[4:]
19  # 0 1 2 3 4 5 6 7 8
20  print(same_values is my_list)   # False
21
22  # both start and end are empty: WHOLE list
23  same_values = my_list[:]
24
25  print(same_values)
```

Slice with a **positive** step

```
4   my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
5
6   sub_list = my_list[1:8  ]    # 1 2 3 4 5 6 7
7   sub_list = my_list[1:8:1]    # 1 2 3 4 5 6 7
8   sub_list = my_list[1:8:2]    # 1 3 5 7
9   sub_list = my_list[1:8:3]    # 1 4 7
10
11  # Missing step: default = 1
12  sub_list = my_list[1:8: ]   # [1, 2, 3, 4, 5, 6, 7]
13
```

- Practice well before next time
- Next
  - Missing values for (start, end, step)
  - Negative step
  - Replace and delete

Slice with a **positive** step

```
4   my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
5
6   sub_list = my_list[1:8 ]    # 1 2 3 4 5 6 7
7   sub_list = my_list[1:8:1]   # 1 2 3 4 5 6 7
8   sub_list = my_list[1:8:2]   # 1 3 5 7
9   sub_list = my_list[1:8:3]   # 1 4 7
10
11  # Missing step: default = 1
12  sub_list = my_list[1:8: ]   # [1, 2, 3, 4, 5, 6, 7]
13
14  # Positive step: Missing end: default is len(seq)
15  sub_list = my_list[1:9:2]   # 1 3 5 7
16  sub_list = my_list[1: :2]   # 1 3 5 7
17
18  # Positive step: Missing start: default is 0
19  sub_list = my_list[0:6:2]   # 0 2 4
20  sub_list = my_list[ :6:2]   # 0 2 4
21
22  sub_list = my_list[0:9:2]   # 0 2 4 6 8
23  sub_list = my_list[ : :2]   # 0 2 4 6 8
24
25  sub_list = my_list[0:9:1]   # [0, 1, 2, 3, 4, 5, 6, 7, 8]
26  sub_list = my_list[ : : ]   # [0, 1, 2, 3, 4, 5, 6, 7, 8]
27
```

Slice with a **negative** step

```
3   my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
4
5   sub_list = my_list[1:8:1]    # 1 2 3 4 5 6 7
6
7   sub_list = my_list[8:1:-1]   # 8 7 6 5 4 3 2: high to low
8
9   sub_list = my_list[7:0:-1]   # 7 6 5 4 3 2 1
10  sub_list = my_list[7:0:-2]   # [7, 5, 3, 1]
11
12  sub_list = my_list[2:5:-1]   # [] must be high to low
13
14  # Negative step: Missing start: default is len
15  sub_list = my_list[9:2:-1]      # [8, 7, 6, 5, 4, 3]
16  sub_list = my_list[ :2:-1]      # [8, 7, 6, 5, 4, 3]
17
18  # Negative step: Missing end: default is hmm
19  # starts from index 0 INCLUSIVE (NOT default)
20  sub_list = my_list[5: :-1]      # [5, 4, 3, 2, 1, 0]
21
22  sub_list = my_list[5:0:-1]      # [5, 4, 3, 2, 1]
23
24  sub_list = my_list[::-1]        # reversed list
25  # [8, 7, 6, 5, 4, 3, 2, 1, 0]
26
```

Slicing with -ve indexing and -ve step

```
2
3           #-9 -8 -7 -6 -5 -4 -3 -2 -1    # 9 + neg_pos
4   my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
5
6   sub_list = my_list[1:8:1]     # 1 2 3 4 5 6 7
7
8   sub_list = my_list[-8:-2:1]    # 1 2 3 4 5 6
9   sub_list = my_list[-2:-8:-1]   # 7 6 5 4 3 2
10
11
```

Replace and Delete

```
2
3     lst = [1, 2, 3, 4, 5, 6, 7]
4     lst[2] = 100    # 1 2 100 4 5 6 7
5
6     lst[3:6] = [982]    # 1 2 100 982 7
7
8     lst[1:3] = [10, 11, 12, 13]    # 1 10 11 12 13 982 7
9
10    # you need to replace 3 times with LIST OF THREE
11    #lst[1:6:2] = [1]   # ValueError
12    lst[1:6:2] = [-1, -2, -3]    # 1 -1 11 -2 13 -3 7
13    #lst[6:2:-2] = [0]   # ValueError
14
15    lst[3:] = [123]    # 1 -1 11 123
16
17    lst = [1, 2, 3, 4, 5, 6, 7]
18
19    del lst[1:3]          # 1 4 5 6 7
20
21    del lst[1:5:2]       # 1 5 7
22
23
```

Python Programming

Tuples

Tuples

- Another an **ordered** collection of objects
  - Some pronounce it as though it were spelled **“too-ple”**
  - and others as though it were spelled **“tup-ple”**
- Several similarities with list
  - Iterating, Indexing, slicing, comparisons, multiple elements: min(), max(), sorted()
- More:
  - A im**mutable** data type: We can’t change or delete ith item
    - Many methods don’t exist: append, insert, remove
    - Though we can change the item internal content if mutable!
  - Fast iteration (visible with large collection)
  - Key with Dict. List can’t
  - Multiple return from a function or multiple assignments

Creation

```
4   t = ('mostafa', 12, 2.5, 12)     # 4 items!
5   t = ('mostafa', 12, 2.5, 12, )   # also 4 items!
6
7   t = (10)
8   print(type(t))  # SADLY int not tuple :(
9   t = (10, )      # tuple of 1 item
10  t = ()          # tuple of 0 item
11
12  print(len((True, 'mostafa')))     # 2
13
14  # all are tuples
15  x, y = 1, 2
16  x, y = (1, 2)
17  (x, y) = (1, 2)
18
19  # TypeError: tuple expected at most 1 arguments, got 3
20  #t = tuple(1, 2, 3)
21  t = tuple((1, 2, 3))     # constructor: iterable
22  t = tuple([1, 2, 3])
23  t = tuple('most')        # ('m', 'o', 's', 't')
24
```

Indexing and Slicing

```
# Same as lists

numbers = (10, 2, 7, 5, 3)

print(numbers[0], numbers[-1])  # 10 3

print(numbers[2:])    # (7, 5, 3)
print(numbers[::])    # (10, 2, 7, 5, 3)
print(numbers[::-1])  # (3, 5, 7, 2, 10)

for item in numbers:
    print(item, end=' ')    # 10 2 7 5 3

#TypeError: 'tuple' object does not support item assignment
#numbers[0] = 4
```

Methods and Functions

```
3   numbers = (10, 2, 7, 2, 2, -5)
4
5   print(numbers.count(2))      # 3
6   print(numbers.index(2))      # 1
7
8   #AttributeError: 'tuple' object has no attribute 'remove'
9   #numbers.remove(0)
10
11  #TypeError: 'tuple' object doesn't support item deletion
12  #del numbers[0]
13
14  print(min(numbers), max(numbers))    # -5 10
15
16  lst = sorted(numbers)    # LIST: [-5, 2, 2, 2, 7, 10]
17
18  print(tuple(sorted(numbers))) #   (-5, 2, 2, 2, 7, 10)
19  print(tuple(reversed(numbers))) # (-5, 2, 2, 7, 2, 10)
20
```

+ And * Operators

```
t1 = (1, 2, 3)
t2 = ('mostafa', True)

t = t1 + 2 * t2

print(t)
# (1, 2, 3, 'mostafa', True, 'mostafa', True)

# TypeError: can only concatenate tuple (not "list") to tuple
#t = t1 + [2, 3, 4]

print(('Hi') * 4)     # HiHiHiHi
print(('Hi',) * 4)    # ('Hi', 'Hi', 'Hi', 'Hi')
```

Comparisons

```
3   # Same rules for comparison as list/string
4
5   t1 = (1, 2, 3)
6   t2 = (1, 2)
7
8   print(t1 < t2)  # False
9
10  print((1, 2) + (3, 4) == (1, 2, 3, 4))  # True
```


Tuples
Unpacking

* For unpacking

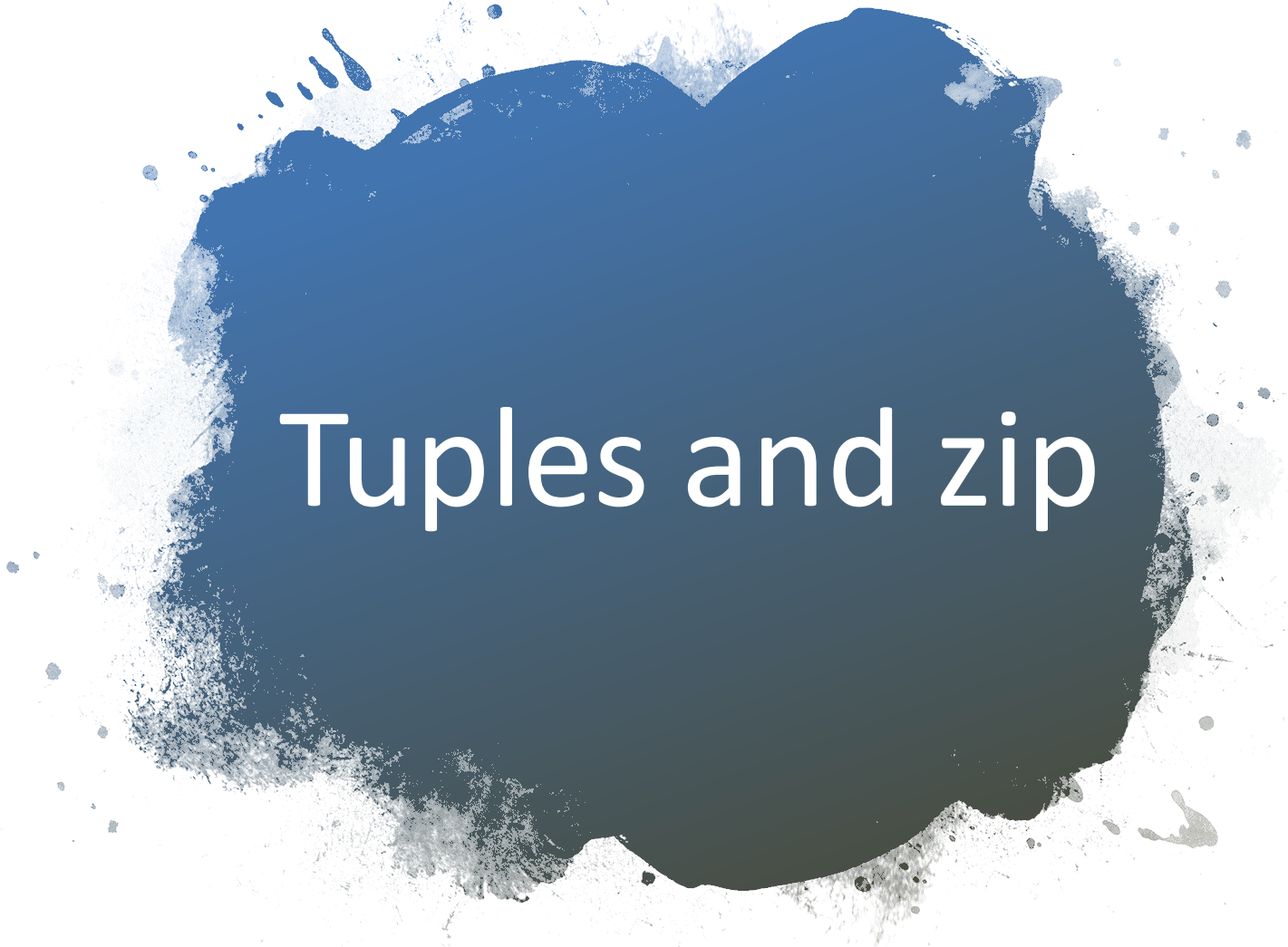
```
lst = 1, 2, 3, 4, 5
a, b, c, d, e = lst      # normal unpacking
a, _, _, _, _ = lst      # what If i don't care? use _ a common notation

# what if I am not sure from the total number? use *
    # * here refers to varying number of arguments
a, b, *c = lst
print(c)    # [3, 4, 5]

*a, b, c = lst
print(a)    # [1, 2, 3]

a, *b, c = lst
print(b)    # [2, 3, 4]

a, *b, c, d = lst
print(b)    # [2, 3]

# Although we can do the same with slicing
# but the * operator is more elegant and makes code simpler!

def f(*items):
    print(items)  # (1, 2, 3, 4)

f(1, 2, 3, 4)
```

Unpacking

```
3   lst = [1, 2, 3]
4   print(lst)      # [1, 2, 3]
5   print(*lst)     # 1 2 3   unpack first, then print: print received 3 arguments NOT 1
6
7
8   def f(a, b):
9       print(a+b)
10
11  #f(*lst)    f() takes 2 positional arguments but 3 were given
12
13  lst1 = [1, 2, 3]
14  lst2 = [4, 5, 6]
15  conc = [*lst1, *lst2]
16  print(conc) # [1, 2, 3, 4, 5, 6]
17
```

Deep unpacking

```
lst = 1, 2, (5, 6)

#ValueError: not enough values to unpack (expected 4, got 3)
#a, b, c, d = lst

print(len(lst)) # 3

# deep unpacking
a, b, (c, d) = lst
print(a, b, c, d)   # 1 2 5 6

t = 1, 2, 3, (4, (5, 6))
a, b, c, (d, (e, f)) = t
print(a, b, c, d, e, f) # 1 2 3 4 5 6
```


Tuples and zip

Zip Function

```
# When you have multiple sequences and want to iterate
# such that in each iteration you have a single item
# from each sequence ==> you need zip

numbers = [1, 2, 3]
letters = ['a', 'b', 'c']

# zip class constructor: def __init__(self, *iterables)
# it takes a group of iterables
# it then returns iterator that we can use to iterate

zipped = zip(numbers, letters)

print(list(zipped))
# [(1, 'a'), (2, 'b'), (3, 'c')]

words = ["mostafa", 'saad', 'ibrahim']
print(list(zip(numbers, letters, words)))
# [(1, 'a', 'mostafa'), (2, 'b', 'saad'), (3, 'c', 'ibrahim')]

# note: zip() in Python 3 is different than Python 2
```

zip

```
2   numbers = [1, 2, 3]
3   letters = ['a', 'b', 'c']
4   words = ["mostafa", 'saad', 'ibrahim']
5
6   for tuple_item in zip(numbers, words, letters):
7       print(tuple_item)
8
9   """
10  (1, 'mostafa', 'a')
11  (2, 'saad', 'b')
12  (3, 'ibrahim', 'c')
13  """
14
15  for number, word, letter in zip(numbers, words, letters):
16      print(number, word, letter)
17  """
18  1 mostafa a
19  2 saad b
20  3 ibrahim c
21  """
```

Different length?

```
3   # what if sequences are of different length?
4   # It stops at the shortest length
5
6   items = list(zip(range(10, 15), range(100)))
7   print(items)
8   # [(10, 0), (11, 1), (12, 2), (13, 3), (14, 4)]
9   # observe stopped only after 5 elements!
10
11  # unzip
12  seq1, seq2 = zip(*items)
13  print(seq1)     # (10, 11, 12, 13, 14)
14  print(seq2)     # (0, 1, 2, 3, 4)
15
16
```

Python Programming

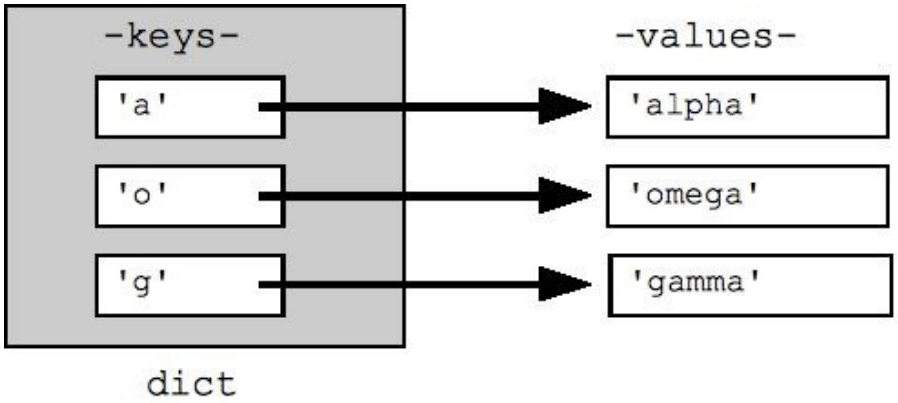
Dictionary

From list to Dict

```
lst = [10, 22, 55]
# idx 0->10  1->22  2->55
print(lst[1])        # 22  [1] is called index
# indices are from 0 to N-1

dict = {0:10, 1:22, 2:55, 12345:37}
print(dict[1])       # 22  [1] is called key
print(dict[12345])   # 37
# Keys are provided: {0, 1, 2, 12345}
# Format {key1:value1, key2:value2, etc}
# Dictionary data structure "associates" key with value

```

Flexible keys

```python
# The key can be from ANY IMMUTABLE value
# This what make dict very useful

dict = {'a' : 'alpha',  # key:value
        'o': 'omega',
        'g': 'gamma'
        }
print(dict)
# {'a': 'alpha', 'o': 'omega', 'g': 'gamma'}
print(dict.keys())
# dict_keys(['a', 'o', 'g'])
print(dict.values())
# dict_values(['alpha', 'omega', 'gamma'])

print(dict['a'])    # alpha
```



Img src

Update and delete

```python
#Dict is mutable. We can update its content
dict = {}    # No initial value
dict[12] = [405, (1, 'mostafa')]    # Add a new key-value
dict['mostafa'] = 20

print(dict[12]) # [405, (1, 'mostafa')]
dict[12] = 'hello'
print(dict[12]) # hello
print(dict.keys())  # dict_keys([12, 'mostafa'])

del dict[12]
print(dict.keys())  # dict_keys(['mostafa'])
#print(dict[12])    # KeyError: 12

dict[12] = 10
dict[12] += 5
print(dict[12])    # 15
print(dict.pop(12)) # 15 : get and remove
print(dict.pop('hey', 37))  # 37 default value
```

Set default

```
dict = {
    int: [6, 9, 10],
    float : 10,
    6: 20,
    6: 70,
    6: 80,
}
print(dict[float])  # 10    observe we can use data types, as they are immutable
print(dict[6])      # 80    mutliple same keys: last value is used
#print(dict[7])     # KeyError: 7

# setdefault: returns the value of the item with the specified key.
# If the key does not exist, insert the key, with the specified value
print(dict.setdefault(6, -8))   # 80
print(dict.setdefault(7, 20))   # 20
print(dict[7])      # 20
```

Membership Operator

```
dict = {
    -1200001 : 'mostafa',
    'ziad' : 25.5,
    (4, 6) : [5, 8, 9],
}

print('ziad' in dict)    # True
print(100 in dict)       # False

#if dict[7] == 5:     # KeyError: 7
#    pass

if 7 in dict and dict[7] == 5:
    pass    # short-circuit evaluation
```

Get method

```
dict = {
    -1200001 : 'mostafa',
    'ziad' : 25.5,
    (4, 6) : [5, 8, 9],
}

print(dict.get(7))          # None
print(dict.get(7, 15))      # 15    (return default val if not exist)
print(7 in dict)            # False
print(dict.get((4, 6)))     # [5, 8, 9]

dict.clear()        # remove all keys

```

Popitem method

```
dict = {'x': 11, 'b': 22, 'y': 30}
dict['a'] = 33

while dict:
    print(dict.popitem())
"""
removes the last key-value pair added from d
    and returns it as a tuple:
('a', 33)
('y', 30)
('b', 22)
('x', 11)
"""
```

Insertion order: NOW preserved (Python 3.7)

```
dict = {}   # empty dict
dict[20] = 10
dict['mostafa'] = 10
dict[30] = 15
dict[(2, 7)] = 150
dict[30] = 10
# observe: values can be anything and can repeat

print(dict.keys())
# dict_keys([20, 'mostafa', 30, (2, 7)])
# Starting from python 3.7 specification : the keys order is preserved (insertion order)
# However, due to several reasons, it is still best practice to not depend on that
# Maybe after 10 years. For now, if order matter: use OrderedDict
# In practice: typically u don't care about insertion order but sorted keys themselves
```

Keys!

```
dict = {'x': 11, 'b': 22, 'y': 30}

print(dict.items())     # dict_items([('x', 11), ('b', 22), ('y', 30)])

for key, value in dict.items():
    print(key, value)    # x 11 b 22 y 30

print(dict.keys())  # dict_keys(['x', 'b', 'y'])
print(list(dict.keys()))  # ['x', 'b', 'y']

for key in dict.keys():
    print(key, dict[key])    # same, but slower (extra access)

for key in sorted(list(dict.keys())):
    print(key, dict[key])    # sorted keys: b x y

for key in sorted(dict):     # shortcut for ordered keys
    print(key, dict[key])  # sorted keys: b x y
```

List vs Dict

```
lst = [10, 20, 30, 40]
print(lst)
# [10, 20, 30, 40]
# list: ordered sequence
# can be indexed or sliced

dict = {0:10, 3:40, 2:30, 1:20}
print(list(dict.values()))
# [10, 40, 30, 20]
# dict: ORDERED collection of key-value-pairs
# items INSERTION order is preserved (3.7)
```

Merge, len, all, any

```
3   dict = {'x': 11, 'b': 22, 'y': 30}
4   dict['a'] = 33
5
6   dict.update({'aaa':3, 'b':-2})  # merge
7   print(str(dict)) # {'x': 11, 'b': -2, 'y': 30, 'a': 33, 'aaa': 3}
8   # you can pass dict or list of tuples
9
10  print(len(dict))    # 5
11
12  # True if all keys are trye
13  print(all(dict))    # True
14  dict[''] = "hey"
15  print(all(dict))    # False
16  print(any(dict))    # True
17
```

fromkeys

```
3   a = [1, 2, 20, 6, 210, 2, 1]
4   d = dict.fromkeys(a)
5   # {1: None, 2: None, 20: None, 6: None, 210: None}
6
7   print(dict.fromkeys(a, 7))
8   # {1: 7, 2: 7, 20: 7, 6: 7, 210: 7}
9
10  unique_keys = dict.fromkeys(a).keys()
11  print(unique_keys)    # dict_keys([1, 2, 20, 6, 210])
12  # removed duplicated + preserved the order
13
14  unique_keys = list({10:2, 1:5})
15  print(unique_keys)  # [10, 1]
```

Python Programming

Set

What?

```
# set: unordered
# (don't preserve insertion order / no values order)
# unique: duplicates are ignored
# items: must be immutable

st = set()
st.add(20)
st.add(10)
st.add(20)
st.add(-2537)
st.add(10)
print(st)   # {10, 20, -2537}
```

Set

```
3   st = {1, 5, 1, 3, 5}
4   print(st)    # {1, 3, 5}
5
6   st = set(['saad', 'most', 'saad'])  # takes iterable
7
8   print('al' in st)          # False
9   for item in st:            # No guarantee on order
10      print(item, end=' ') # most saad
11  print()
12
13  print(list(st))       # ['most', 'saad']
14  print(set({1:10, -2:30}))    # {1, -2}
15
16  print(set('Hey'))          # {'H', 'y', 'e'}
17  print(set(['Hey']))        # {'Hey'}
18  print(set({'Hey'}))        # {'Hey'}
```

dic

Functions

```
st = {(1, 5), (2, 7), (1, 5), (2, 7)}
print(st)    # {(2, 7), (1, 5)}

# TypeError: unhashable type: 'list'
#st = {(1, 5), [2, 7]}

print(len(st))  # 2
print(max(st))  # (2, 7)
print(sorted(st))  #[(1, 5), (2, 7)]

print(sum({1, 1, 1, 1, 2, 2, 2, 2}))    # 3 = 1+2
print(all({1, 2, 'hey'}))       # True
print(all({1, 2, 'hey', ()}))   # False: empty tuple
```

Methods

```
3   st1 = {1, 3, 5, 7, 8, 10}
4
5   st1.add(-20)
6   st1.remove(10)
7   #st1.remove(30)  # if not exist, error => KeyError
8   st1.discard(30)  # if not exist, no problem
9   print(st1)  # {1, 3, 5, 7, 8, -20}
10
11  print(st1.pop())    # remove random element. If empty = error
12  st1.clear() # remove elements
```

Union and Intersection

```
st1 = {1, 5, 7, 8}
st2 = {1, 5, 3, 10}

print(st1 | st2)          # {1, 3, 5, 7, 8, 10}: union using | operator
print(st1.union(st2))     # same
print(st1.union([1, -5, -7]))     # pass any iterable
# note: st1 is not updated

st3 = {5, 6, 1}
su = st1 | st2 | st3
si = st1 & st2 & st3      # set intersection
print(si)     # {1, 5}
print(st1.intersection(st2).intersection(st3))   # {1, 5}
print(st1.intersection(st2, st3))   # {1, 5}

```

Difference

```
st1 = {1, 5, 7, 8}
st2 = {1, 5, 3, 10}

# return the set of all elements that are in st1 but not in st2
print(st1 - st2)                    # {8, 7}
print(st1.difference(st2))          # same

# return the set of all elements in either st1 or st2, but not both:
print(st1 ^ st2)          # {3, 7, 8, 10}
print(st1.symmetric_difference(st2))

# True if no intersection
print(st1.isdisjoint(st2))          # False
print(st1.isdisjoint([4, 6]))       # True
```

Is subset? superset?

```
2   st1 = {1, 5}
3   st2 = {2, 1, 5, 3}
4
5   # True if every element of st1 is in st2
6   print(st1 <= st2)          # True
7   print(st1.issubset(st2))   # True
8
9   # True if every element of st1 is in st2, but not equal
10  print(st1 < st2)           # True
11  print(st1 < {1, 5})        # False
12
13  print(st2 >= st1)          # True
14  print(st2.issuperset(st1)) # True
15  print(st1 >= {1, 5})       # True
16  print(st1 > {1, 5})        # False
```

Updates

```
st1 = {1, 5, 7, 8}
st2 = {1, 5, 3, 10}

st1 |= st2  # union and update st1
st2.update(st1)

# same &= ^=
```

frozenset

```
# immutable set
st1 = frozenset([7, 5, 1, 8])
# can't change it: no add/remove etc

print(id(st1))  # 0x111
st1 |= {20, 10}
print(id(st1))  # 0x222 DIFFERENT - recall strings!

# useful if u need a set, but immutable
dct = {st1 : 5}

for item in sorted(st1):
    print(item, end=' ')
    # 1 5 7 8 10 20
```