SELA

Module 2: Python basic types

Agenda

- Python data types
- Names of Python identifiers
- Boolean and Numeric Variables
- Basic Numeric Operators
- Math — Mathematical functions
- Strings Variables
- String Operators and Built-in Methods
- Combine Statements
- Python Sequences

Agenda

- Python Lists, Operators and Methods
- Iterating Lists and lists comprehension
- Python Tuple
- Python Dictionary
- Iterating Through a Dictionary and dictionary comprehension
- Python Set
- Data Type Conversion
- Mutable vs Immutable in Python

Python data types

- Python is dynamic language, so you don't need to declare your variables
- Variables must be assign before we can use them
- Python built-in types:
    - Boolean
    - None
    - Numeric types
        - Integers - equivalent to C longs
        - float: Floating-Point numbers, equivalent to C doubles
        - complex: Complex Number
    - Sequences:
        - String
        - List
        - Tuple
        - Set
        - Dictionary

Names of Python identifiers

- Rules for a python identifiers (variable name, function, class, module):

  - A identifier name may contain:

    - Digits (0 to 9).
    - Letters, both lower case and upper case ('a'-'z' and 'A'-'Z').
    - Underscores ('_').

  - The first character must not be a digit and preferable not underscore

  - It is extremely important to choose meaningful names for identifiers and not preserved python name.

  - All identifier names are case-sensitive.

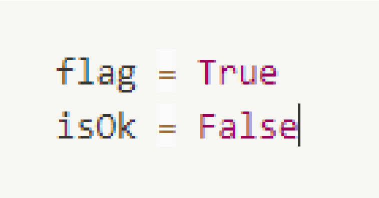
For example: **MyClass**, **PersonModel**, **CarFactory**.

Names of Python identifiers – cont’d

- Naming convention for Python:
  - All identifiers should be lowercased with underscore as words separato: (*sum_of_digits*)
  - Class names start with an uppercase letter and all other identifiers, like variables and functions with a lowercase letter : `MyClass` , `PersonModel`, `CarFactory`.
  - Starting an identifier with a single leading underscore indicates, by convention, that the identifier is meant to be private : `_internal_variable`, `_private_method`
  - If the identifier starts and ends with two trailing underscores, the identifier is a language-defined special name: `__init__`, `__name__`

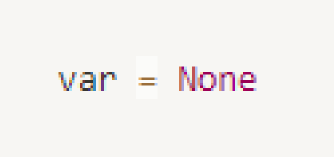
Boolean Variables and None

- Boolean variables:
  - Boolean variables can hold only True or False values.
  - For example:



```
flag = True
isOk = False
```

- None value (of NoneType)



```
var = None
```

None is used to represent the absence of a value

Numeric types

- In python 2.X integer can be represented by both *int* and *long* types. *int* is for small values (4 bytes), *long* have unlimited digits count.
- In python 3.x there is only one integer type – *int, with* unlimited digits count (line *long* in python 2.x).

int: 123, -78, 0x5A, 0567

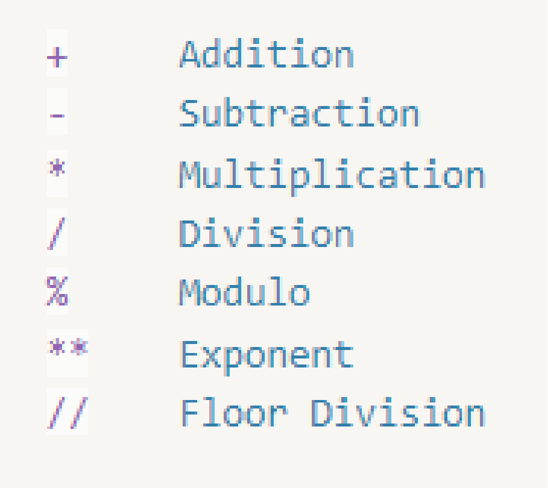
long: 51924361987678, 0xDEFABCECBDAECBFBAEl

float: 0.5, -77.99, 12.3+e45

complex: 3.14j, -0.6545+0J, 4.53e-7j

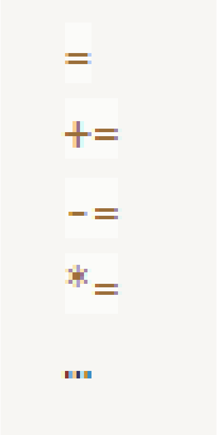
Basic Numeric Operators

- Arithmetic



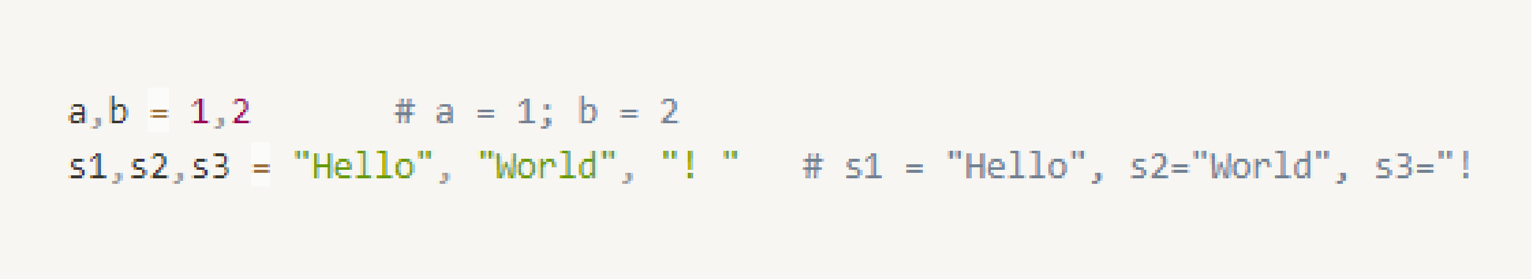
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
**	Exponent
//	Floor Division

- Assignment



```
=
+=
-=
*=
...
```

- Corresponding assignment (value objects count must be equal to value count)



```
a,b = 1,2        # a = 1; b = 2
s1,s2,s3 = "Hello", "World", "! "   # s1 = "Hello", s2="World", s3="!
```

Numeric Operators

Demo

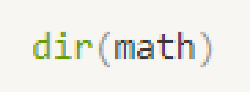
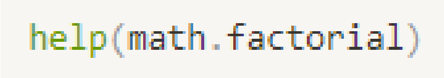
- Example 1:

```
n1 = 8
n2 = 7
res = n1+n2 * 2   # res = 22
res -= 10          #res = 12
```

- Example 2:

```
num1 = 8; num2 = 3; num3 = 8.0
res1 = int(num1/num2)   #  res1 = 2
res2 = num3/num2        #  res2 = 2.667
res3 = num1//num2       #  res3 = 2
```

Math — Mathematical functions

- We can *import math* module to use mathematics functions (will be discussed later in the course)
- *math* module contains functions like: pow, sqrt, exp, log, sin, etc
- To see all math definitions use:  `dir(math)`
- To see the documentation about some function of *math* module use built-in help function:  `help(math.factorial)`

For example:

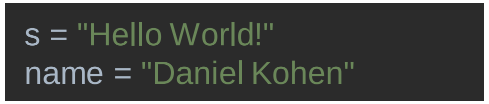
```
import math
num = math.sqrt(16)
print(num)      #prints 4
```

Python index-base Sequences

- The most basic data structure in Python is a **sequence**.
- Python has several built-in types of sequences: lists, tuples, dictionaries, sets and strings
- The lists, tuples and strings are index-based sequences.
- The index-based sequences are support index access and index-range access
- positive and negative indexes are supported (positive indexes starts with 0, negative with -1)
    - A negative indexes are counted from the end. So we can access the last element is with -1 index, the second to the last element would be -2, and so on.

Strings Variables

- Strings in Python can be created using single quotes, double quotes, and triple quotes.
- Python treats single quotes and double quotes strings the same:
- For example:



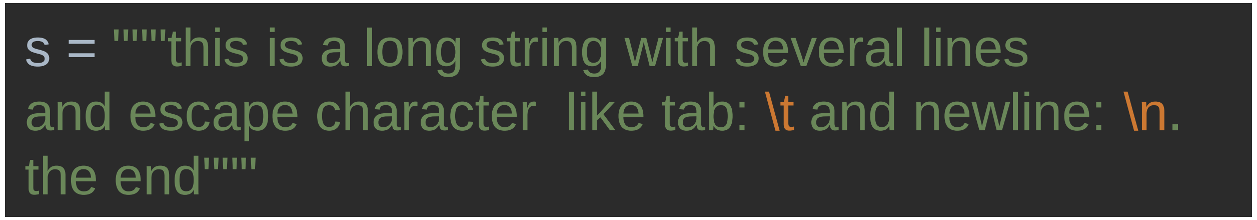
```
s = "Hello World!"
name = "Daniel Kohen"
```

- An escape character interpreted in both single-quoted and double-quoted strings.

Strings Variables – triple quotes strings

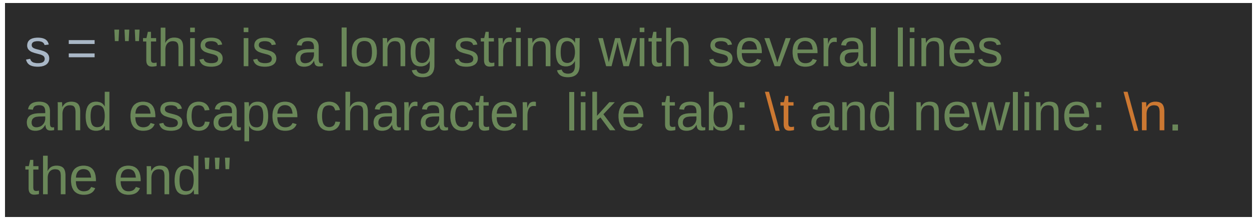
- Triple quotes strings can span for several lines and they consist of three consecutive single or double quotes.

- For example:



```
s = """this is a long string with several lines
and escape character  like tab: \t and newline: \n.
the end"""
```

Or:

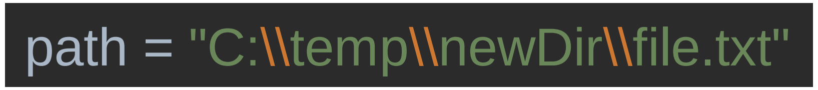


```
s = '''this is a long string with several lines
and escape character  like tab: \t and newline: \n.
the end'''
```

Strings Variables – raw strings

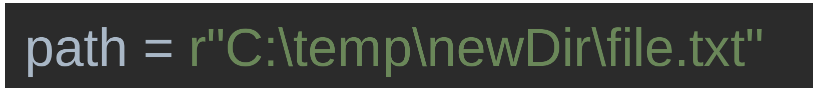
- Python can define raw strings, that blocks escape characters
- Putting the ***r*** character before single, double or triple quotes string will define them as raw-string
- Raw-strings are used to backslash escape characters

- For example:



```
path = "C:\\temp\\newDir\\file.txt"
```

– double back-slash for each back-slash character
Or

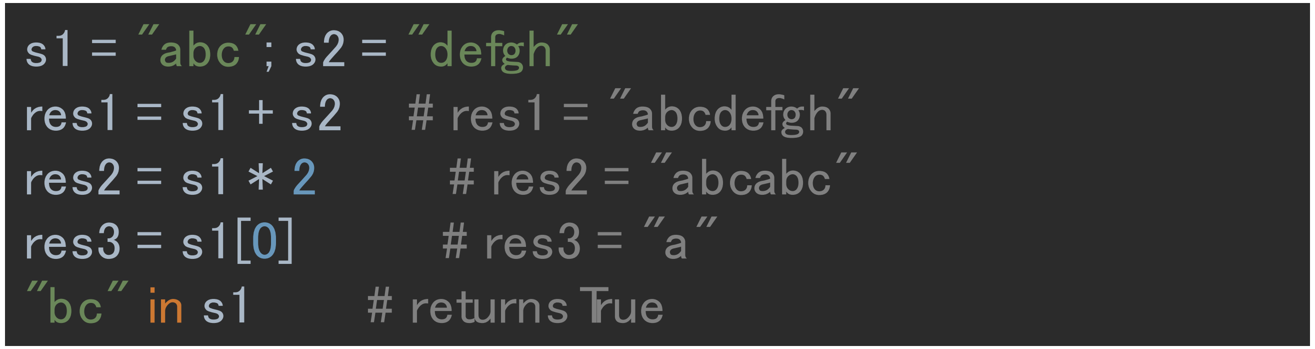


```
path = r"C:\temp\newDir\file.txt"
```

String Operators

+	Concatenation
*	Repetition
[]	Slice
[:]	Range Slice
in/not in	Membership
%	Format

- <u>**Example :**</u>



```
s1 = ″abc″; s2 = ″defgh″
res1 = s1 + s2    # res1 = ″abcdefgh″
res2 = s1 * 2       # res2 = ″abcabc″
res3 = s1[0]        # res3 = ″a″
″bc″ in s1        # returns True
```

String ranges

```
s = “abcdef”
print(s[1:4])    # “bcd”
print( s[:4])    # “abcd”
print( s[1:])    # “bcdef”
print( s[1:–1])      # “bcde”
print( s[:])     # “abcdef”
print( s[::–1])      # “fedcba”
print( s[::2])   # “ace”
print( s[1::2])      # “bdf”
```

String as Input

- In Python, the input() function is used to read user input from the standard input (usually the keyboard). It allows the program to pause and wait for the user to enter a value or a line of text.

```
name = input(”Enter your name: ”)
print(”Hello, ” + name + ”!”)
```

Immutability of strings

- Python strings are immutable
- Direct assignment to its items is not supported

```
s = ″abc″
s[0] = ″a″ – ERROR
```

- There are a lot of functions for replacing, sub-stringing, etc. They all built and return the changed string

```
s = ″abac″
   s1 = s.replace(″a″, ″A″)
   print(″s = {}, s1 = {}″.format(s, s1))  #s = ″abac″,
s1=″AbAc″
```

String Built-in Methods

- There are many built-in string functions in python, here some the most common:
- Case functions:
  - **s.capitalize()** - Capitalizes first letter of string
  - **s.lower(), s.upper()** - returns the lowercase /uppercase version of the string
- Test functions:
  - **s.islower(), s.isupper()** - Returns true if all cased characters in string are lowercase/uppercase and false otherwise
  - **s.isalpha(), s. isalnum(), s.isdigit(), s.isspace()...** - tests if all the string characters are in correct state
  - **s.startswith(suffix[, beg=0, end=len(string)])**
  - **s.endswith(suffix[, beg=0, end=len(string)])** - tests if the string starts/ends with the given *suffix*

String Built-in Methods

Demo

Example 1:

```
s  = ″abcd″
if s.islower():
s = s.capitalize()          # s = ″Abcd“
```

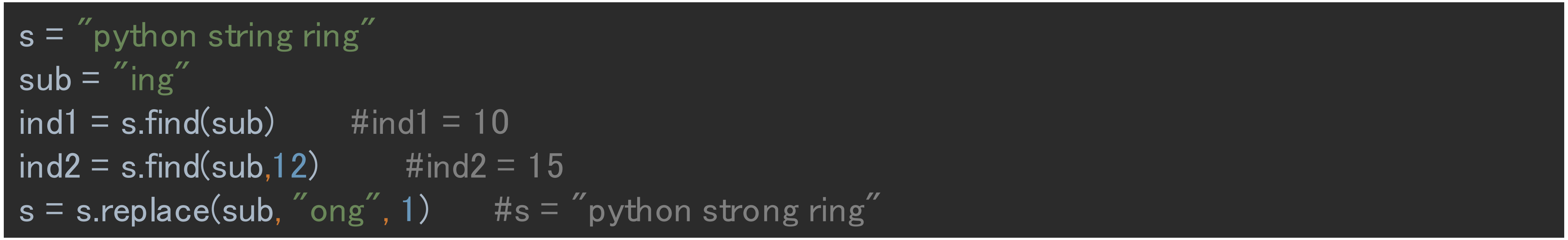
Example 2:

```
if s.startswith(″ab″):
   print( ″starts with ab!″)      # starts with ab!
if s.startswith(″ab″,1,len(s)):
   print( ″starts with ab!″)      # prints nothing, the                # condition does not match
```

String Built-in Methods – cont’d

- Search and replace functions:
  - **s. find(substr[, beg, end=len])/s.rfind** - searches for *substr* in given string s and returns start index of the first/last appearance ,-1 if not found
  - **s.replace(old, new[, max])** - returns a string where all/max occurrences of *old* have been replaced by *new*

- Example:



```
s = ″python string ring″
sub = ″ing″
ind1 = s.find(sub)        #ind1 = 10
ind2 = s.find(sub,12)        #ind2 = 15
s = s.replace(sub, ″ong″, 1)       #s = ″python strong ring″
```

String Built-in Methods – cont’d

- Mix functions:
  - **s.count(str, [beg,end])** - Counts how many times *str* occurs in string
  - **s.strip()** - returns a string with whitespace removed from the start and end
  - **s.split(str [, num])** - Splits string according to delimiter *str* (space if not provided) and returns list of substrings; split into at most num substrings if given
  - **s.join(seq)** - Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string

String Built-in Methods – cont’d

- Example:

```
s = ″python+strings+example″
sub = ″n″
cnt = s.count(sub)        #cnt = 2
lst = s.split(″+″)       #list = [″python″,″strings″,″example″]

str =″−″
lst = [″a″,″b″,″c″,″d″]
str = strjoin(lst)          #″a−b−c−d″
```

String format function

- Use format function to build formatted strings
- Curly-brackets are place holders. They are zero based

```
n1,n2 = 4,9
print("{0} + {1} = {2}".format(n1,n2, n1+n2))       # 4 + 9 = 13

print("{1} + {0} = {2}".format(n1,n2, n1+n2))       # 9 + 4 = 13
```

- Starting from python 2.7, the indexes can be omitted

```
print("{} + {} = {}".format(n1,n2, n1+n2))                # 4 + 9  = 13
```

- The same item can have multiple references

```
print("{0} is {0}".format(n1)         # 4 is 4
```

String format function – cont’d

- Place holders can be named

```
print("first value is {first_val}, second value is {second_val}".format(
    second_val=1, first_val=2))             #first value is 1, second value is 2
```

- Width and alignment can be specified

```
print("{0:<20}.".format("A"))          # A                   .
print("{:^20}.".format("A"))           #          A          .
print("{:>20}.".format("A"))           #                    A.
```

- Floating point precision can be specified

```
res = 10.0/3
print("{:.2f}".format(res))            # 3.33
```

Python Built-in function

- Python has a few built-in function, some of them work on sequences
- len(seq) – returns number of items in sequence-*seq*
- max(seq) – returns an item with maximal value in sequence-*seq*
- Etc

- Unpack assignment is also supported on sequences:

```
s = "klm"
c1,c2,c3 = s    # c1 = "k", c2 = "l", c3 = "m"
```

Combine Statements

- Multi-Line Statements:
  - Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the command should continue. For example:

```
total = item_one + \
            item_two + \
            item_three_statements
```

  - Statements contained within the [], {} or () brackets do not need to use the line continuation character. For example:

```
days = ["Monday", "Tuesday", "Wednesday",
 "Thursday", "Friday"]
```

- Multiple-statements in one line
  - Use ";" to separate multiple statements in one code line.

```
n1 = 9; n2 = 11; n3 = -6
```

Python Lists

- The list is created using the square brackets [].

```
actors = ["Jack Nicholson", "Antony Hopkins",
 "Adrien Brody"]
```

- Python lists are mutable

- Python lists can hold mixed types

```
list = [11, "hello", -3.14, 23]
```

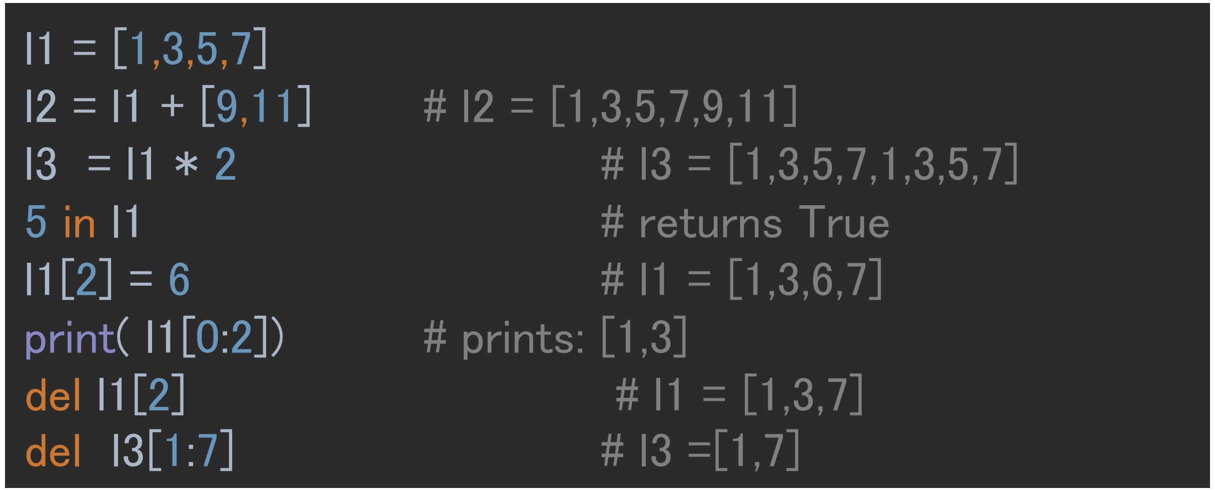
- List items can be accessed by index:

```
first_name = actors[0]          # first_name = "Jack Nicholson"

last_name = actors[-1]          # lst_name = "Adrien Brody"
```

Python Lists Operators

+	Concatenation
*	Repetition
[]	Slice
[:]	Range Slice
in/not in	Membership
del	Delete List Elements Or entire list



```
l1 = [1,3,5,7]
l2 = l1 + [9,11]        # l2 = [1,3,5,7,9,11]
l3  = l1 * 2                  # l3 = [1,3,5,7,1,3,5,7]
5 in l1                       # returns True
l1[2] = 6                     # l1 = [1,3,6,7]
print( l1[0:2])         # prints: [1,3]
del l1[2]                      # l1 = [1,3,7]
del  l3[1:7]                  # l3 =[1,7]
```

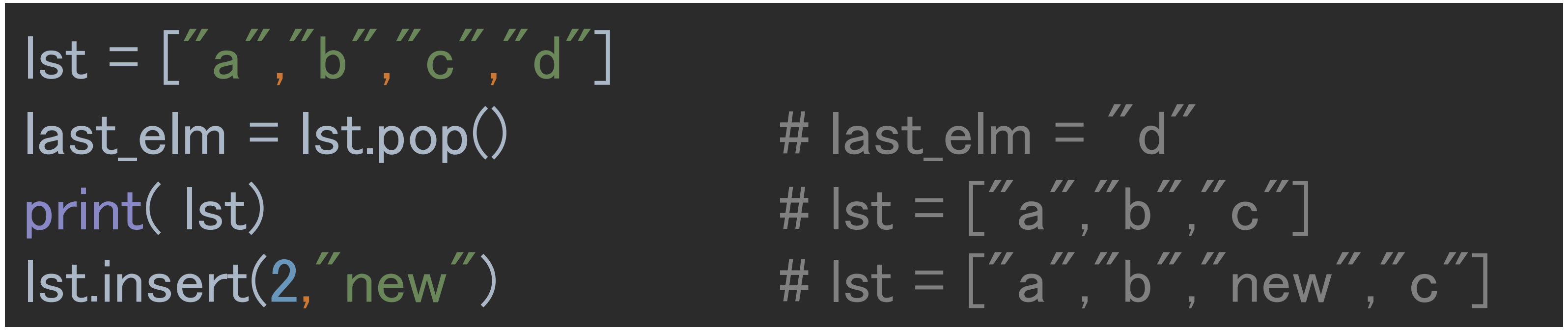
Python Lists Operators – cont'd

```
values = [1, 3, 5, 7, 9, 11, 13]
print( values[:])      # prints all, like print(values)
print( values[1:])        # prints all except first element
print( values[:1] )       # prints only first element
print( values[2:4])       # prints values at index 2 and 3

del values[:]      # deletes all list values, values = []
del values     # undefined list, list doesn’t exists now
print( values) # generates an error, the list is              # longer exists
```

Python Lists Methods

- Add/ Remove elements:
  - **list.append(obj)** – appends *obj* to the list
  - **list.extend(seq)** – appends *seq* to the list
  - **list.insert(index, obj)** – inset *obj* to the list at *index* index
  - **list.pop()** - removes and returns last object from list
  - **list.remove(obj)** – removes *obj* from list

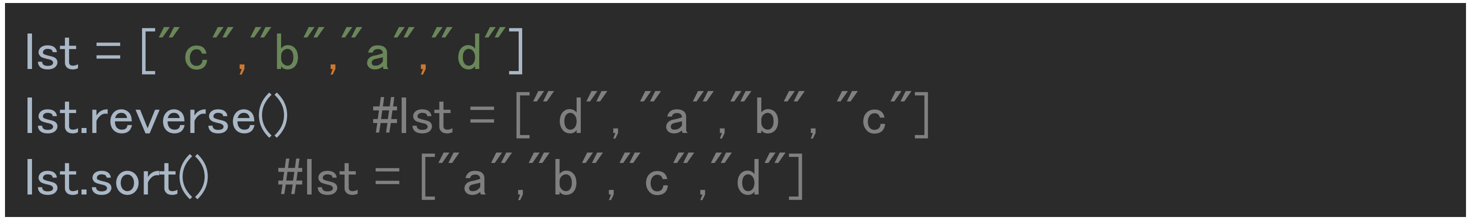


```
lst = [”a”,”b”,”c”,”d”]
last_elm = lst.pop()          # last_elm = ”d”
print( lst)                   # lst = [”a”,”b”,”c”]
lst.insert(2,”new”)           # lst = [”a”,”b”,”new”,”c”]
```

Python Lists Methods – cont'd

- <u>Reorganize functions:</u>
  - **list.sort()** – sorts list items, IN PLACE
    - Sort function can receive call-back to specify custom sort order and attributes (will be discussed later)
  - **list.reverse()** – reverse the order of list items, IN PLACE

  - **list.index(obj)** – returns first index of obj in list, -1 otherwise
  - **list.count(obj)** – returns the number of times obj appears in list
- Example:



```
lst = [”c”,”b”,”a”,”d”]
lst.reverse()     #lst = [”d”, ”a”,”b”, ”c”]
lst.sort()    #lst = [”a”,”b”,”c”,”d”]
```

Iterating Lists

- Iterating values

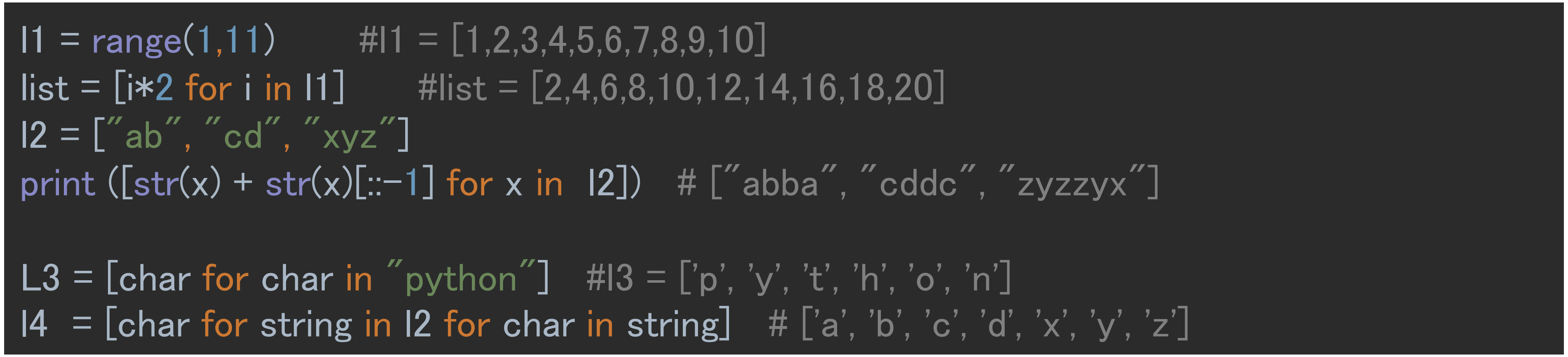
```
for elm in li:
        print( elm)
```

- Iterating indexes:

```
li = ['a', 'b', 'c', 'd', 'e']
for i in range(len(li)):
        print( li[i])
```

Lists comprehension

- Lists comprehension used to construct new list from existed sequence in a very natural, easy way



```
l1 = range(1,11)       #l1 = [1,2,3,4,5,6,7,8,9,10]
list = [i*2 for i in l1]      #list = [2,4,6,8,10,12,14,16,18,20]
l2 = [″ab″, ″cd″, ″xyz″]
print ([str(x) + str(x)[::−1] for x in  l2])   # [″abba″, ″cddc″, ″zyzzyx″]

L3 = [char for char in ″python″]   #l3 = [’p’, ’y’, ’t’, ’h’, ’o’, ’n’]
l4  = [char for string in l2 for char in string]   # [’a’, ’b’, ’c’, ’d’, ’x’, ’y’, ’z’]
```

Python Tuples

- A tuple is index based sequence, just like list.
- Python tuples are enclosed in parentheses ( )
- Tuples are immutable and cannot be updated.
- Tuples can be thought of as read-only lists
- Tuples they more effective that lists

- For example:

```
tuple = ( "abcd", 786 , 2.23, "john", 70.2 )
```

Tuples can’t be changed

```
tup = ("a", "b", "c", "d")
```

```
del tup[1]              # generates an error
tup[0] = "e"            # generates an error
```

But reassignment is supported:

```
tup = (1, 2, 3)         #correct
```

Tuples Operators

- Tuples have the same operators as lists and they behave the same:

+, *, [], [:], in/not in, Unpack assignment

Example 1:

```
tup = (1, 2, 3, 4, 5, 6, 7 )
res_tup = tup[1:5]    # res_tup = (2,3,4,5)
```

Example 2:

```
tup1 = (”12”, ”234”)
tup2 = (”34”, ”567”, ”8”)
tup3 = tup1 + tup2    # tup3 = (”12”, ”234”, ”34”, ”567”, ”8”)
```

Python Dictionary

- Python's dictionaries are kind of hash table that can be found in lots of different programming languages
- Dictionaries consist of key-value pairs.
- Dictionaries are enclosed in curly braces { }
- Dictionary values have no restrictions. They can be any built-in or user-defined type
- There are two important points to remember about dictionary keys:
  - Keys must be immutable (for built ins) or hashable (for user-defined)
  - no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins
- A dictionary is mutable type

Python Dictionary – cont'd

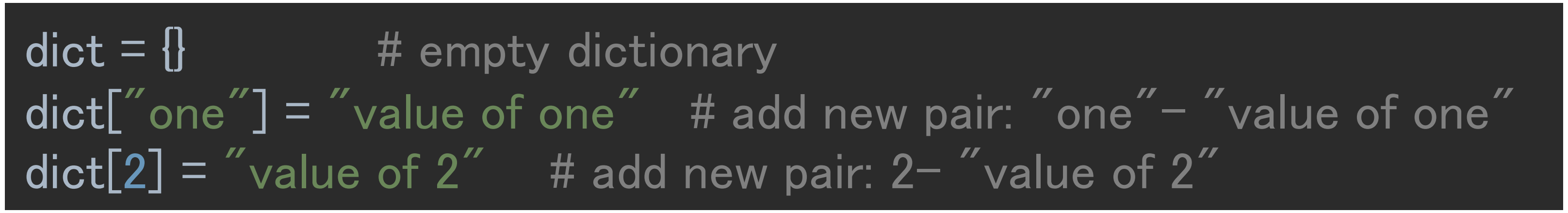
- Python dictionaries have they own internal keys organization, this is why the order of its items(pairs) is unpredictable and should be treated like "random"

```
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
print(dict)                #prints dict = {'Age': 7, 'Name': 'Manni'}
```

- Values can be assigned and accessed using square braces [] for keys
    - When non-existed key is assigned with a value, this key-value pair is added to dictionary
    - When existed key is assigned with a value, its value is updated
    - When access non-existed key, the error is generated

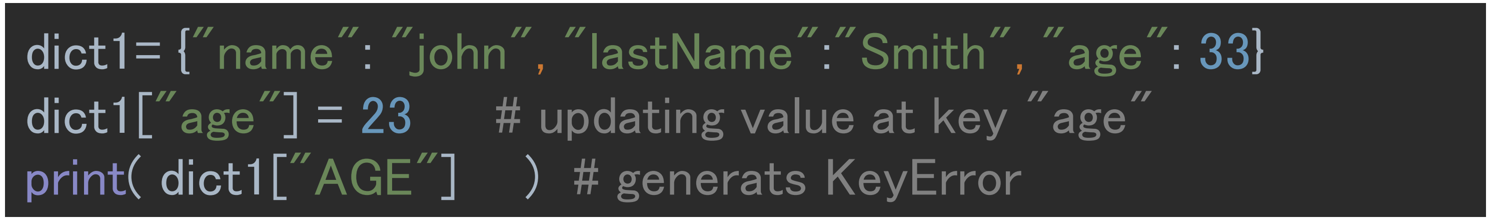
Python Dictionary – examples

- Create and Updating dictionary
  - Way 1: step by step creation



```
dict = {}              # empty dictionary
dict[″one″] = ″value of one″    # add new pair: ″one″− ″value of one″
dict[2] = ″value of 2″     # add new pair: 2− ″value of 2″
```

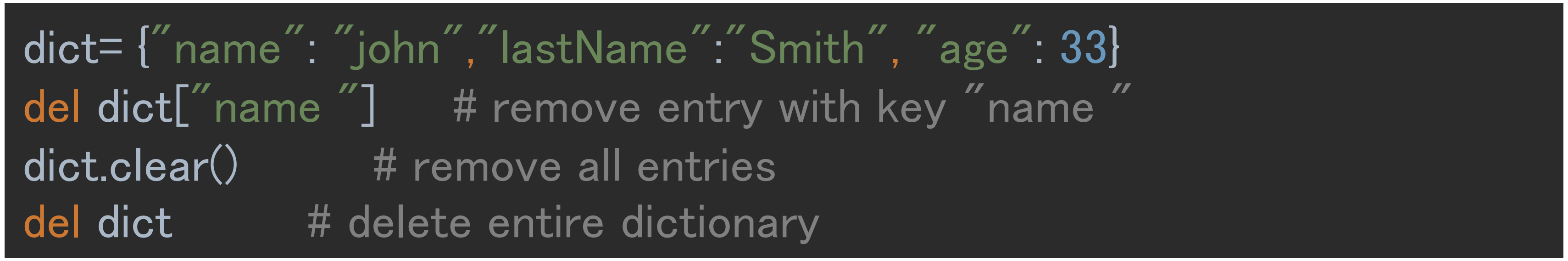
  - Way 2: all at once creation



```
dict1= {″name″: ″john″, ″lastName″:″Smith″, ″age″: 33}
dict1[″age″] = 23      # updating value at key ″age″
print( dict1[″AGE″]    ) # generats KeyError
```

Python Dictionary – cont'd

- Delete Dictionary Elements:
- You can either remove individual dictionary elements, clear the entire contents of a dictionary or delete entire dictionary in a single operation.



```
dict= {″name″: ″john″,″lastName″:″Smith″, ″age″: 33}
del dict[″name ″]      # remove entry with key ″name ″
dict.clear()          # remove all entries
del dict          # delete entire dictionary
```

Python Dictionary – cont'd

- The following methods defined for dictionary:

  len(dic) – returns the number of key-value pairs

  dict.clear() - removes all elements of dictionary *dict*

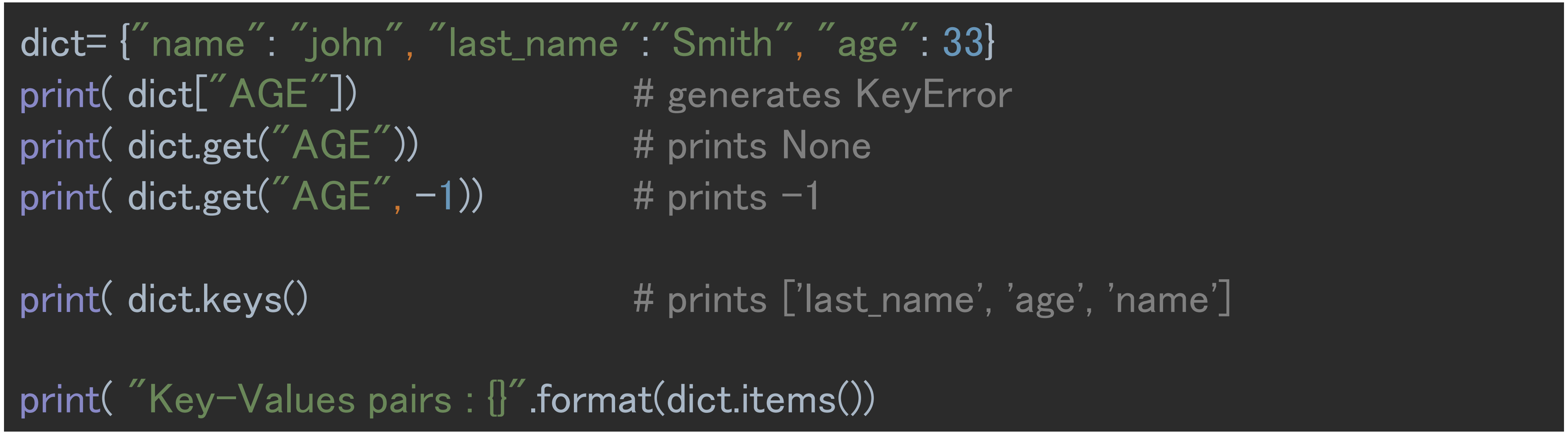
  dict.get(key,[default for non-existed key]) - for *key* key, returns its value or *default* if *key* doesn’t not exists. Default value for *default* is None

  dict.items() -returns a list of *dict*'s (key, value) tuple pairs

  dict.keys() -returns list of dictionary *dict's* keys

  dict.values() -returns list of dictionary *dict*'s values

Python Dictionary – cont'd



```
dict= {″name″: ″john″, ″last_name″:″Smith″, ″age″: 33}
print( dict[″AGE″])                 # generates KeyError
print( dict.get(″AGE″))             # prints None
print( dict.get(″AGE″, −1))         # prints −1

print( dict.keys()                  # prints [’last_name’, ’age’, ’name’]

print( ″Key−Values pairs : {}″.format(dict.items())
```

The Output:

Key-Values pairs : [('last_name', 'Smith'), ('age', 33), ('name', 'john')]

Iterating Through a Dictionary

- d = {'x': 1, 'y': 2, 'z': 3}

- Iterating keys:

```
for key in d.keys():
        print( key, d[key])
```

- Iterating items:

```
for k, v in dict.items():
     print( "{}, {}".format (k, v))
```

The Output:

```
y, 2
x, 1
z, 3
```

Iterating Through a Dictionary – cont’d

- Iterating values:

```
dict= {"name": "john", "last_name":"Smith", "age": 33}
```

```
print( "\n".join(["%s=%s" % (k, v) for k, v  in dict.items()]))
```

The output:

```
Last_name=Smith
age=33
name=john
```

Dictionary comprehension

- Dictionary comprehension supported in python starting from python 2.7

```
keys = [1,2,3]
values=[4,5,6]
dict = {keys[i]:values[i] for i in range(len(keys))}
print(dict)                                       # {1: 4, 2: 5, 3: 6}

d1 = {k: 0 for k in ['a','b','c']}               # {'a': 0, 'c': 0, 'b': 0}
d2 = {n: n**2 for n in [10, 11, 12]}         #{10:100, 11:121, 12:144}
```

Set’s functions

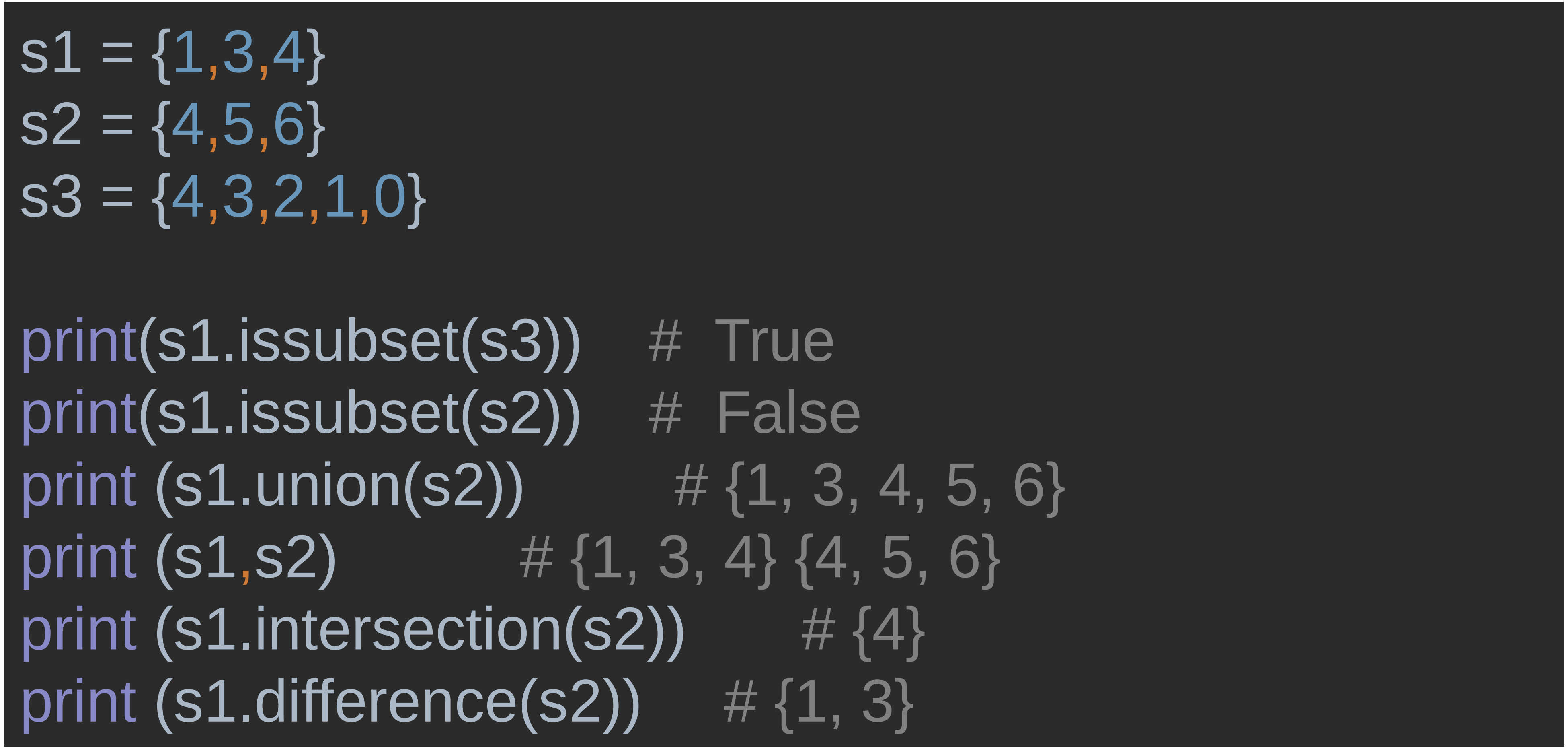
st.issubset(t)	test whether every element in st is in t
s.issuperset(t)	test whether every element in t is in st
s.union(t)	return new set with elements from both s and t
s.intersection(t)	return new set with elements common to s and t
s.difference(t)	return new set with elements in s but not in t
s.symmetric_difference(t)	return new set with elements in either s or t but not both
s.update(t)	update set s with elements added from t
s.difference_update(t)	update set s after removing elements found in t
s.add(x)	add element x to set s
s.discard(x)	removes x from set s if present
s.clear()	remove all elements from set s

Set’s functions examples

```
items = set()
items.add("cat")
items.add("dog")
items.add("gerbil")
print(items)      # {'gerbil', 'dog', 'cat'}

s1 = {1,2,4,5,6}
s2 = {2,3,5,7,8}
s2.update(s1)
print (s2)          #{1, 2, 3, 4, 5, 6, 7, 8}
```

Set’s functions examples – cont’d



```
s1 = {1,3,4}
s2 = {4,5,6}
s3 = {4,3,2,1,0}

print(s1.issubset(s3))    #  True
print(s1.issubset(s2))    #  False
print (s1.union(s2))        # {1, 3, 4, 5, 6}
print (s1,s2)           # {1, 3, 4} {4, 5, 6}
print (s1.intersection(s2))      # {4}
print (s1.difference(s2))     # {1, 3}
```

Mutable vs Immutable in Python

- An immutable variable is an object whose value can't be modified after it is created
- If change of immutable variable is supported, the change will always create a new, updated value
- Python immutable type are: None, bool, int, long (if exist), complex, string, tuple
- Python mutable types are: list, dictionary, set, classes

Mutable vs Immutable in Python – cont’d

- Look at the following code:

```
l1 = [11, 22, 33]
l2 = l1;
l1.append(44)
print( l2)
```

What is the output for this code?

Variable Definition

Demo

Labs 01-02

Lab

Questions ?