Python Lecture 2 – Data in python

- Data types

S. Bertocco

Bibliography and learning materials

- ★ Bibliography:
  https://www.python.org/doc/
  http://docs.python.it/
  https://www.w3schools.com/python/
  and much more available in internet

- ★ Learning Materials:
  https://github.com/gtaffoni/Learn-Python/tree/master/Lectures

  https://github.com/bertocco/abilita_info_units_2021

Data types

Python has two families of data types:

Simple data types:

– Int

– Float

– Complex

– Boolean

– String

Container data types:

– list []

– tuple ()

– dict {}

– set

– frozenset

Simple Data Types

– Int

– Float

– Complex

– Boolean

– String

Numeric types

In python there are 3 numeric types:

- Integer
- Float
- Complex

In general, an n-bit integer has values ranging from -2^(n-1) to 2^(n-1) – 1

information about integer dimension:

```
>>> sys.maxsize
9223372036854775807
```

information about the internal representation of floating point

```
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53,
        epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Numeric types: Examples

int type can be:

in base 2 (using the prefix 0b)	bin(19)
in base 10,	
in base 16 (using the prefix 0x)	hex(300)
in base 8 (using the prefix 0)	oct(300)

```
>>> a=300
>>> oct(a)
'0454'
>>> hex(a)
'0x12c'
>>> bin(a)
'0b100101100'
>>> bin(19)
'0b10011'
```

Numeric types

float are real number in double precision.

Examples:

```
>>> a = 12.456
>>> c = 12232e-2
>>> b = .2
>>> a=6.12244e-5
>>> type(a)
<class 'float'>
```

Note: floor (troncamento) function is the function that takes as input a real number x and gives as output the greatest integer less than or equal to x.
floor: 2.1 → 2 -0.1 → -1

The ceiling (arrotondamento) function maps x to the least integer greater than or equal to x
ceiling: -0.99 → 0 2.1 → 3

Be careful using int and float:

What happens doing...

100/3	division int/int
100//3	floor division int/int (gets the integer part)
100.0/3	division float/int
100.0//3	floor division float/int
100%3	remainder of the division int/int
divmod(100,3)	The divmod() method takes two numbers and returns a pair of numbers (a tuple) consisting of their quotient and remainder.

Numeric types

Complex Number represents a complex number in double precision.
The real and the imaginary parts can be accessed using the functions ‘real’ and ‘imag’.

Example:

```
>>> r=12+5j              # ‘j’ symbol means the imaginary part
>> r=10+5j
>>> type(r)
<class 'complex'>

>>> r.real
12.0
>>> r.imag
5.0
>>> type(r.real)
<type 'float'>
>>> type(r.imag)
<type 'float'>
```

Operations on numeric types

In Python operations on numeric types are managed by the following operators:

- Arithmetic operators
- Comparison operators
- logical operators
- bitwise (bit a bit) operators
- membership operators
- identity operators

Some built-in functions working on numeric data:
- abs(number) returns the absolute value of a number
- pow(x, y[, z]) returns the value of x to the power of y (x^y). If a third parameter is present, it returns x to the power of y, modulus z.
- round(number[, ndigits]) returns a floating point number that is a rounded version of the specified number, with the specified number of decimals.

Executing operations between different numeric type variables, the implicit conversion rule is:

$$\text{Int} \longrightarrow \text{Float} \longrightarrow \text{Complex}$$

Arithmetic operators

a=10   b=21

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a+b=31
- Subtraction	Subtracts right hand operand from left hand operand.	a-b=-11
* Multiplication	Multiplies values on either side of the operator	a*b=210
/ Division	Divides left hand operand by right hand operand	b/a=2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b%a=1
** Exponent	Performs exponential (power) calculation on operators	a**b=10^{20}
// Floor division	The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity)	9//2 = 4 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

https://www.tutorialspoint.com/python3/python_basic_operators.htm

Comparison operators

a=10   b=20

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) is true
>	If the value of left operand is greater than the value of right operand, then condition becomes true. (a > b) is not true	(a > b) is not true
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true

https://www.tutorialspoint.com/python3/python_basic_operators.htm

Bitwise Operators

Bitwise operator performs bit-by-bit operation.
Examples:

a = 60 and b = 13 in binary format they will be
a =   0011 1100
b =   0000 1101

a&b = 0000 1100   a and b

a|b =   0011 1101   a or b (o uno o l'altro o entrambi)

a^b = 0011 0001   a xor b (o uno o l'altro, non entrambi)

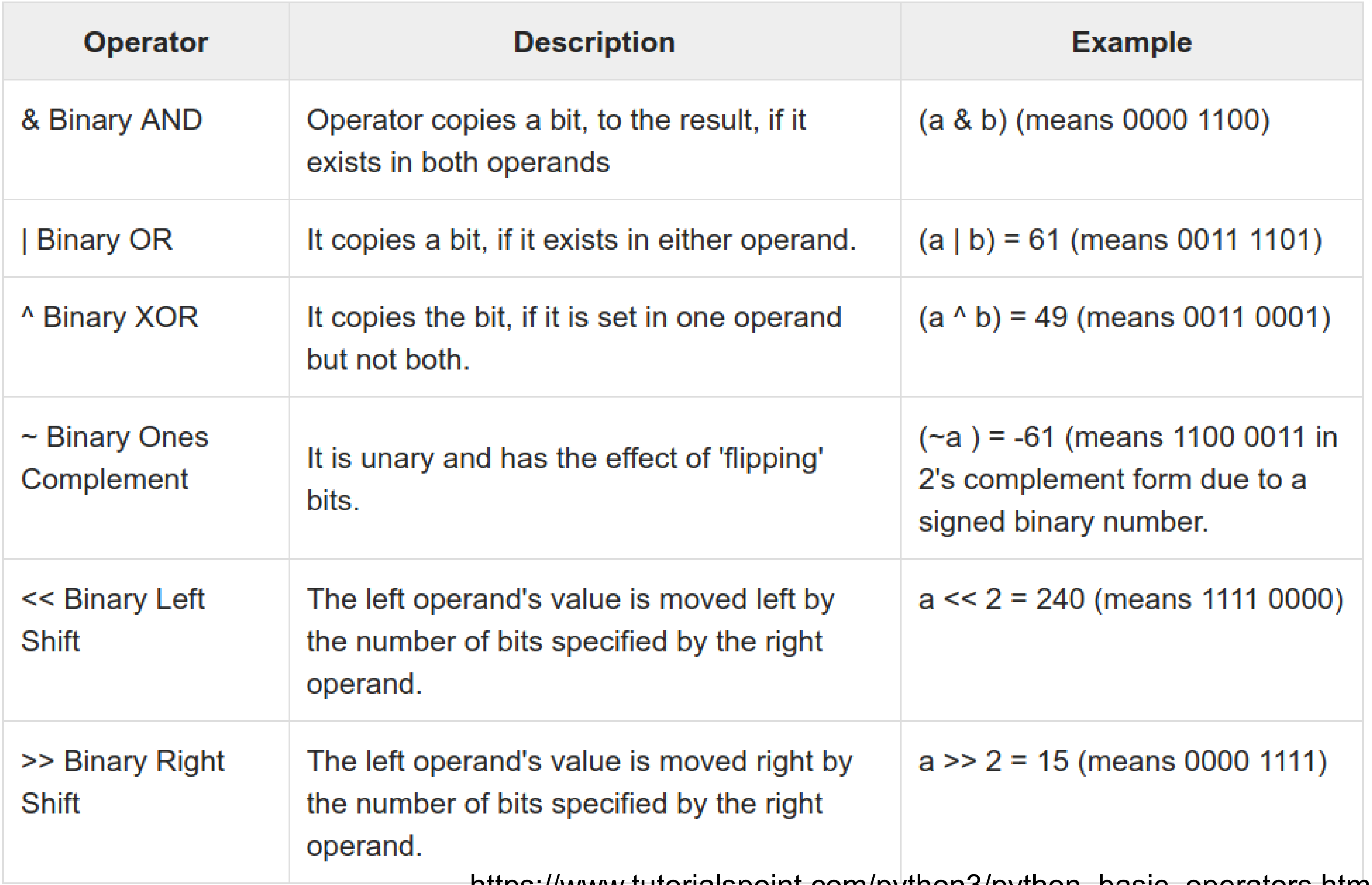
~a =   1100 0011 complement (1→0 and 0->1)

a << 2 = 1111 0000 left shift
a >> 2 = 0000 1111 right shift

Python's built-in function bin() can be used to obtain binary representation of an integer number.

https://www.tutorialspoint.com/python3/python_basic_operators.htm

Bitwise Operators

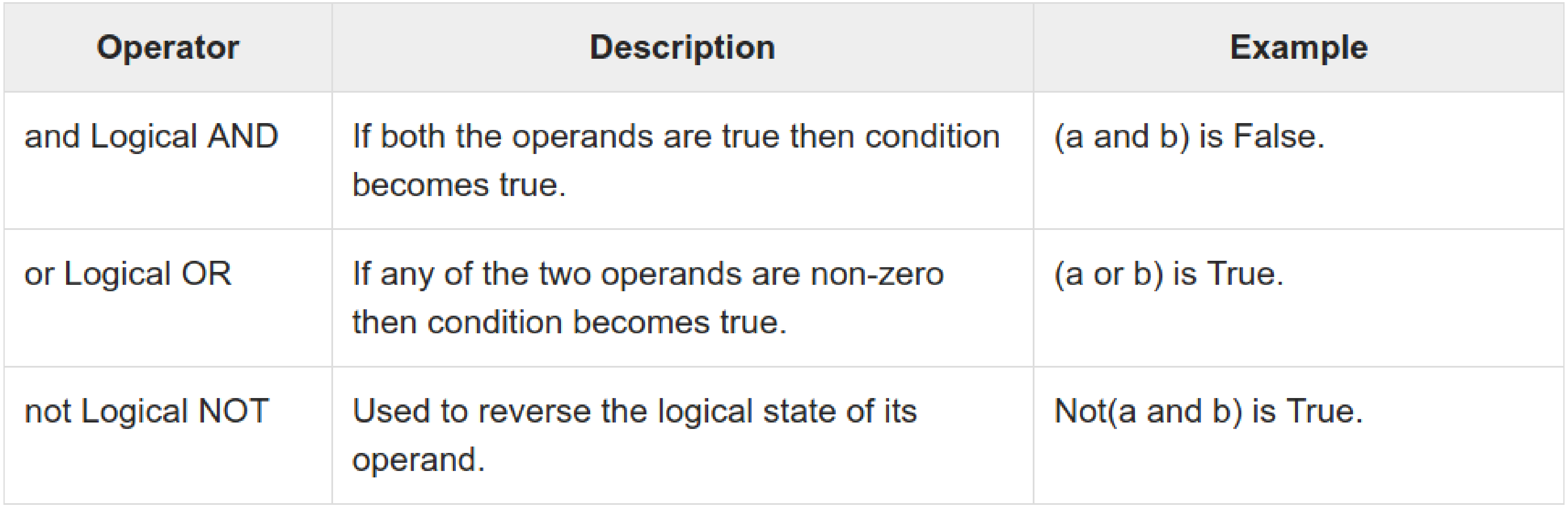


Operator	Description	Example
& Binary AND	Operator copies a bit, to the result, if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit, if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operand's value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

https://www.tutorialspoint.com/python3/python_basic_operators.htm

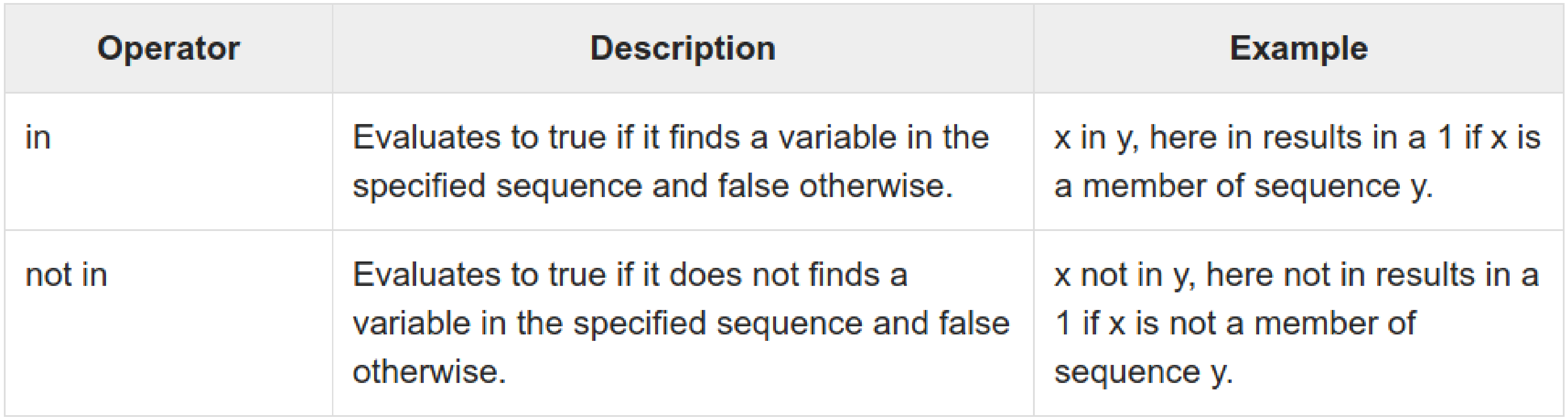
Logical Operators

a = True
b = False



Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is True.

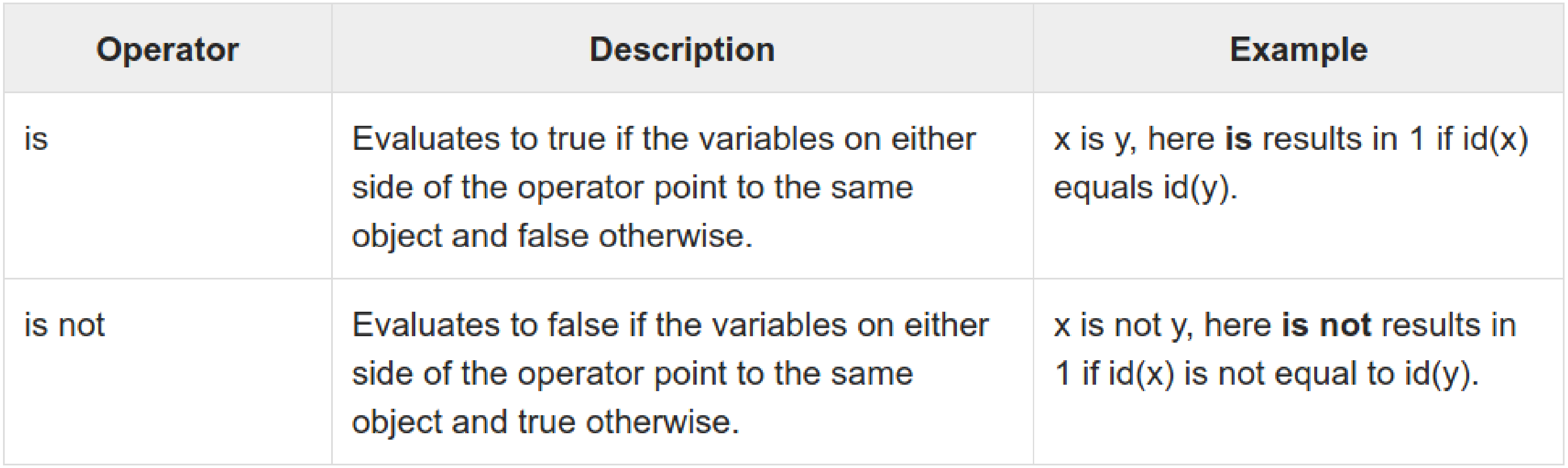
Membership Operators



Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

https://www.tutorialspoint.com/python3/python_basic_operators.htm

Identity Operators



Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

https://www.tutorialspoint.com/python3/python_basic_operators.htm

Examples: Operations on numeric types

```
>>> k=5
>>> s=5+1j              # imaginary part cannot be only j
>>> type(s+k)           # imaginary number conversion
<type 'complex'>

>>> 4 and 2             # logical comparison
2

>>> 4 & 2               # bitwise comparison between the binaries 100 and 010
0

>>> 4 | 2               # bitwise comparison between the binaries 100 and 010
6
```

Operations on numeric types: math module

The math module provides some of the more commons mathematical operations.
It does not work with complex numbers.
cmath module works for complex numbers.

The available functions are:

- Trigonometric functions: cos, sin, tan, asin, acos, atan, sinh, cosh, tanh.
- Exponentiaiton and logarithmic functions: pow, exp, log, log10, sqrt
- Angles representation and conversions: degrees, radians, ceil , floor, fabs

In the math module are defined the numerical constants pi and e

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

Bool type

Booleans True and False are available in Python.
bool is a subclass of int
True corresponds to 1
False corresponds to 0

Integer values can be used to reprsent boolean values with the following convention:
0 corresponds to False
all integer values greater than zero correspond to True

It is good practice to use the bool type to represent boolean values.

Example:

```
>>> a=1
>>> type(a)
<type 'int'>
>>> if(a):
print('True')
True
>>> a=False
>>> type(a)
<type 'bool'>
```

String type

Literal strings are character sequences enclosed in quotes, single or double. Creating strings is as simple as assigning a value to a variable.

Sequences of triple ‘double quotes’ or triple ‘single quotes’ can be used to assign strings spanning in more than one row or containing single or double quotes of the other type.

Example 1:

```
>>> a="""I am a string spanning in 3 rows,
... containing 'sigle quotes',
... containing "double quotes",
... containing '''triple quotes'''
... """
>>> print(a)
I am a string spanning in 3 rows,
containing 'sigle quotes',
containing "double quotes",
containing '''triple quotes'''
```

Example 2:

```
>>> b='''I am a string spanning in 3
rows,
... containing 'sigle quotes',
... containing "double quotes",
... containing """triple quotes"""
... '''
>>> print(b)
I am a string spanning in 3 rows,
containing 'sigle quotes',
containing "double quotes",
containing """triple quotes"""
```

String type

- We can create strings by enclosing characters in quotes (single or double).

Creating strings is as simple as assigning a value to a variable.

Example:

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

- To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

Example:

```
#!/usr/bin/python3
var1 = 'Hello World!'
var2 = "Python Programming"

print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
```

Output:

```
var1[0]:  H
var2[1:5]:  ytho
```

String type

- The type char does not exists. A single char can be accesed using the operator [] or slicing the string with the operator [begin:end] (slicing)

Example:

```
>>> a = “Hello world”
>>> a[1]
‘e’
>>> a[1:2]
‘e’
```

- The single char can not be accessed, but a new value can be assigned to the string

Example:

```
>>> a=‘Primo valore’
>>> a = “Prima valore”     # Ok string re-assignment
>>> a[4] = ‘o’ #Errore      # NOT Ok single character assignment
  File "<stdin>", line 1
    a[4] = ‘o’
         ^
SyntaxError: invalid syntax
```

String type: operators

- The operators + and * can be used for string operations.

Operators priority is maintained.

Example

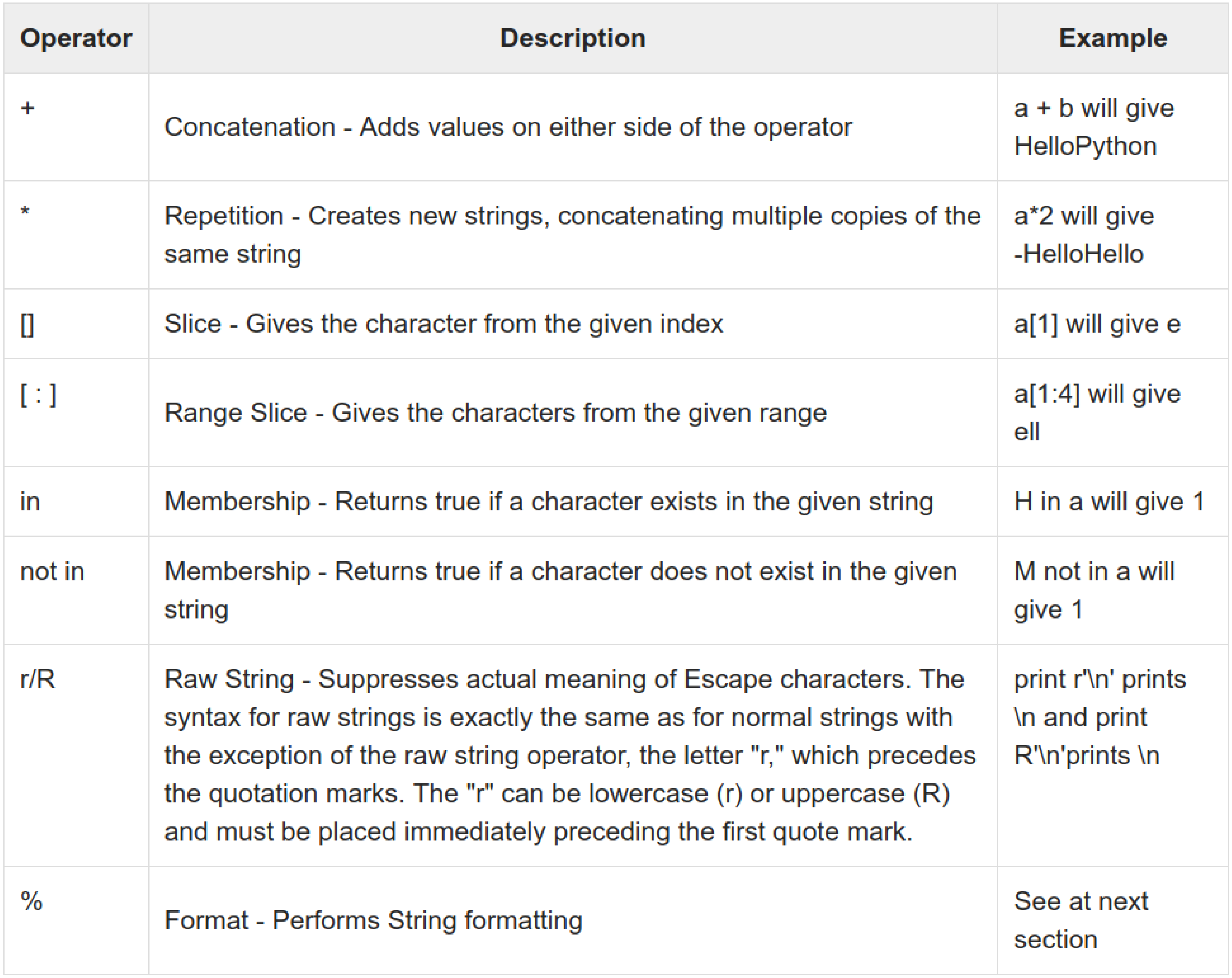
```
>>> a = ‘Hello’
>>> a+a+a                # Concatenation
‘HelloHelloHello’

>>> a = ‘He’+’l’*2+’o World’          # Multiple concatenation
>>> a
‘Hello World’
```

- It exists also the possibility to insert wherever in the string using the operator %

Example

```
name = "Peter"
my_string = "Hello %s" % name                          # Append or insert
my_string = "Hello %s, how are you? %s" % (name, 'ok')      # Insert multiple values
```

String type: operators



Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

String type: escape characters

Escaping allows to add special characters inside a string.

Example

```
>>> a = ‘What’s your name?’ #Errore
SyntaxError: invalid syntax
>>> a = “What’s your name?” # Ok if I create the string with double quotes
>>> a = ‘What\’s your name?’ # Ok if you escape the single quote character
```

Most common escape characters in string manipulation:

• \t Tab	‘Ciao\tciao!’	Ciao ciao!
• \n New Line	‘Ciao\nciao!’	Ciao ciao!
• \\ Backslash	‘c:\\Programmi\\pp’	c:\Programmi\pp
• \” Double quote	‘Repeat: \”Hello\”’	Repeat: ”Hello”
• \’ Single quote	“Repeat:\’Hello\’”	Repeat: ‘Hello’

String type: row string

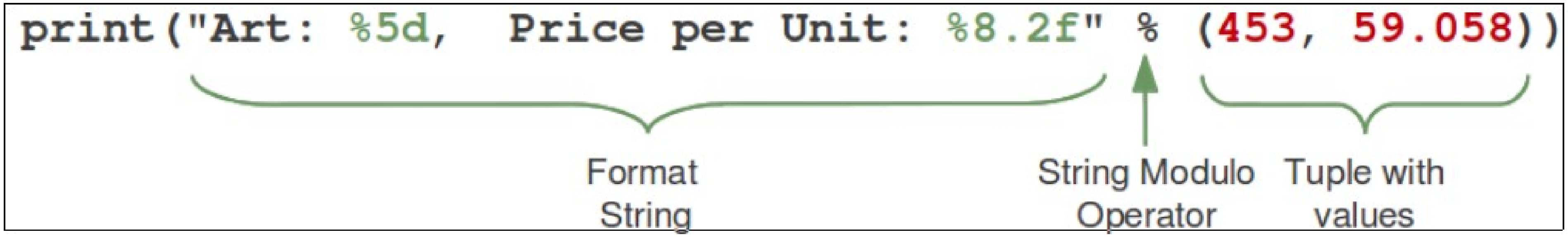
Row string is a string preceded by r or R in front of it.
In a row string a character preceded by \ is included without changes.

Esempio

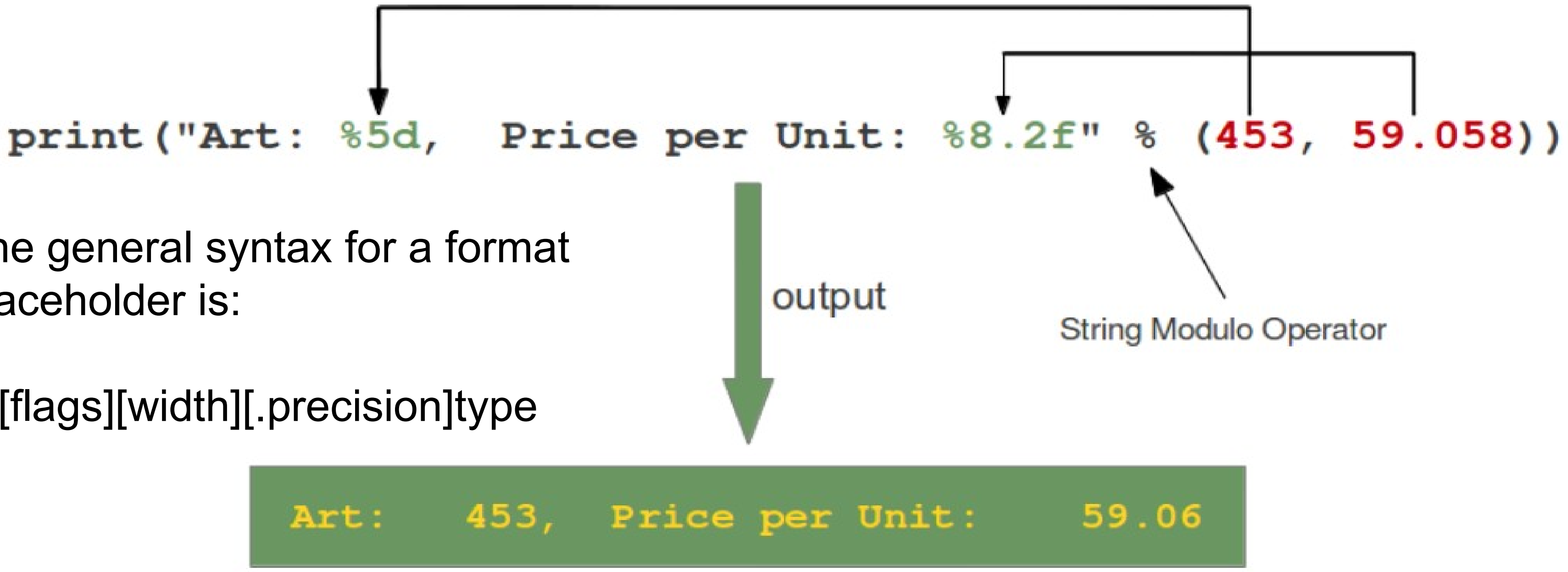
```
>>> a = r'Hello \t World'
#Raw string
>>> a
'Hello \t World'
```

String type: format output

Python allows output formatting.
The % charactes has a special meaning when used in strings, because it is used to format output.



```
print("Art: %5d,  Price per Unit: %8.2f" % (453, 59.058))
```



```
print("Art: %5d,  Price per Unit: %8.2f" % (453, 59.058))
```

```
Art:   453,  Price per Unit:    59.06
```

The general syntax for a format placeholder is:

%[flags][width][.precision]type

String type: format output

Ther are a lot of possibility to format output.

Examples:

%d or %i	Integer
%s,	String
%f,	Floating point decimal format
%c,	Single character (accepts integer or single character string)
%x,	Unsigned hexadecimal (lowercase)
%o,	Unsigned octal
%%,	No argument is converted, results in a "%" character in the result
%e,	Floating point exponential format (lowercase)

Format output: https://www.python-course.eu/python3_formatted_output.php
https://www.tutorialspoint.com/python3/python_strings.htm

String type: format output

Example

```
>>> ”Oggi è %s %d %s” % (“Venerdì’’,20,”Febbraio”)
Oggi è Venerdì 20 Febbraio
```

Already seen example:

```
name = "Peter"
my_string = "Hello %s" % name                    # Append and Insert
my_string = "Hello %s, how are you? %s" % (name, 'ok')      # Insert multiple values
print(my_string)
```

With dictionaries:

```
person = {"name": "John", "age": 19}
print(f"{person['name']} is {person['age']} years old.")
```

# f-string
# full Python expressions inside the braces.

String type: built-in functions

Strings, as all the python objects, have a set of functionalities accessible with built-in Python functions (i.e. functions always available in the python interpreter).

• Manipulate: concat, split, characters deletion and unions.

-split([sep [,maxsplit]])
-replace (old, new[, count])
-strip([chars])

Example
Split:
```
>>> s='Ciao Mondo'
>>> s.split('o',1)
['Cia', ' Mondo‘]
```
Replace:
```
>>> s.replace('o','i',1)
'Ciai Mondo‘
```
Strip:
```
>>> s.strip('C')
'iao Mondo '
```

String type: format built-in functions

Formattazione: align, upper case, lower case

-upper() e lower() e swapcase()

-center(width[, fillchar]) e ljust(width[, fillchar]) e rjust(width[,fillchar])

Example

```
>>> s = ‘Hello’

>>> s.center(10,’.’)
‘..Hello...’

>>> s.upper()
‘HELLO’
```

String type: search built-in functions

- find(sub [,start [,end]])	
- rindex(sub [,start [,end]])	returns the highest index of the substring inside the string (if found). If the substring is not found, it raises an exception.
- index(sub [,start [,end]])	
- rfind(sub [,start [,end]])	returns the highest index of the substring (if found). If not found, it returns -1.
- count(sub[, start[, end]])	
- isupper()	returns whether or not all characters in a string are uppercased or not.
- islower()	returns whether or not all characters in a string are lowercased or not.
- startswith(prefix[, start[, end]])	
- endswith(prefix[, start[, end]])	

Container Data Types

- list []
- tuple ()
- dict {}
- set
- frozenset

list[]

A list is initialized putting elements comma separated inside squared brackets.

- Items in a list can be of different type, both built-in and user defined.
- Indexes in a list start from zero.
- A list can be instantiated without specifying the list length or data type.

Single list elements can be accessed with the operator [ ]

Example:

```
>>>l=[]                # empty list instance
>>> print(l)
[]
>>>m=['Lista','di',4,'elementi']     # initialize a list
>>>print m[2],m[0]                    # access single list elements
>>>4 Lista
```

A list is an ordered sequence list, so the list items order is maintained.

list[] : Basic List Operations

Lists respond to the operators
+ concatenation
* repetition
like strings, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we saw on strings.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

list[] : slicing operator

List support the slicing operator [start:stop:step]

```
L = ['spam', 'Spam', 'SPAM!']
L[2]   ──────►  SPAM!              # Offsets start at zero
L[-2]  ──────►  Spam               # Negative: count from the right
L[1:]  ──►['Spam', 'SPAM!']        # Slicing fetches sections
```

Example:

```
>>>a=[0,1,2,3,4,5,6,7]
>>>a[0:6]
[0,1,2,3,4,5]
>>>a[1:6:2]
[1,3,5]
>>>a[1::2]         # no ‘stop’ means until the end of list
[1,3,5,7]
>>>a[::2]          # no ‘start’ means from the first item of the list
[0,2,4,6]
Slicing can be also negative
>>>a[6:0:-2]       # starts fom index 6, ends to index 0 going back with step 2
[6,4,2]
```

list[] : range() built-in function

The range() function is used to generate lists of integer numbers.

Syntax:
range(start,stop,step) generates a list of integer from ‘start’ to ‘stop’ with interval ‘step’

Example
```
>>>a=range(3)
[0,1,2]

>>>type(a)
<type 'list'>

>>>a=range(1,10)
[0,1,2,3,4,5,6,7,8,9]

>>>a=range(1,10,2)
[1,3,5,7,9]
```

list[] : Complex Operations

Lists supports complex operations.

Examples:

```
>>> a=range(10)
>>> b=[el*2 for el in a]
>>> b
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> l=[1,2]
>>> l2=['a','b']
>>> l3=[4,5]
>>> f=[(e1,e2,e3) for e1 in l for e2 in l2 for e3 in l3]
>>> f
[(1, 'a', 4), (1, 'a', 5), (1, 'b', 4), (1, 'b', 5), (2, 'a', 4), (2, 'a', 5), (2, 'b', 4), (2, 'b', 5)]
```

This can be done also with:

```
>>> for e1 in l:
        for e2 in l2:
            for e3 in l3:
                f.append((e1,e2,e3))
```

list[] : main functions

Function	**Description**
cmp(list1, list2)	Compare elements of lists
len(list)	Gives the total length of the list
max(list)	Returns item from the list with max value
min(list)	Returns item from the list with min value
list(seq)	Converts a tuple into list

Syntax : cmp(list1, list2)

Parameters :
list1 : The first argument list to be compared.
list2 : The second argument list to be compared.

Returns : This function returns 1, if first list is “greater” than second list, -1 if first list is smaller than the second list else it returns 0 if both the lists are equal.

list[] : main methods

List containers can be modified.
List objects contain built-in methods to modify members of a list.

Method	Description
list.append(object)	Appends object to list
list.insert(index, object)	Inserts object obj into list at offset index
list.extend(seq)	Appends the contents of seq to list
list.pop(index)	Removes and returns last object or obj at index from list
list.remove(obj)	Removes object obj from list
list.count(value)	Returns count of how many times value occurs in list
list.index(obj)	Returns the lowest index in list where obj appears
list.reverse()	Reverses objects of list in place
list.sort([func])	Sorts objects of list, use compare func if given

list[] : main methods exercises

Practice with the list methods proposed in the previous slide

list[] : about efficiency

The operators concatenation + (or +=) and repetition * are supported by lists.
The operator + and the function extend() have the same functionality, but different execution time (efficiency)

Example:

```
import time
l=range(100000000)
v=range(1000000)
T1=time.clock()
s=l+v
T2=time.clock()
print(‘ + execution time: :’ , T2-T1, ‘s’)
, “s”
T3=time.clock()
l.extend(v)
T4=time.clock()
print(‘extend execution time:’, T4-T3 , ‘s’)
```

Output:

```
 + execution time: 2.81 s
 extend execution time: 0.033 s
```

list[] for queue and stack

List can be easily used as stack or queue.
pop and append methods can be used to implement the LIFO logic typical of stacks.
pop with index 0 and append can be used to implement the FIFO logic typical of queue.

Example:

```
stack=[1, 2, 3, 4]
print('Initial Stack : ', stack)
for i in range(5,7):
    stack.append(i)
print ("Append: ", stack)
stack.pop()
print ("Pop: " , stack)

queue=[ 'a','b','c','d' ]
print("Initial Queue : ", queue)
queue.append('e')
queue.append('f')
print("Append : ", queue)
queue.pop(0)
print("Pop : ", queue)
```

Output:

```
Initial Stack : [1, 2, 3, 4]
Append: [1, 2, 3, 4, 5, 6]
Pop: [1, 2, 3, 4, 5]
Initial Queue :  ['a', 'b', 'c', 'd']
Append: ['a', 'b', 'c', 'd', 'e', 'f']
Pop: ['b', 'c', 'd', 'e', 'f']
```

tuple()

A tuple is a sequence ordered data enclosed between ().
Tuples are sequences, just like lists. The differences between tuples and lists are,

- the tuples cannot be changed unlike lists, tuple are immutable.
- tuples use parentheses, whereas lists use square brackets.

A tuple is created putting in it different comma-separated values. Optionally, can be put these comma-separated values between parentheses also.
Example:

```
tup1 = "a", "b", "c", "d";
tup2 = ('physics', 'chemistry', 1997, 2000);    # Data in a tuple can be heterogeneous
tup3 = (1, 2, 3, 4, 5 );
```

The empty tuple is written as two parentheses containing nothing. **Example:**

```
tup1 = ();
```

A tuple containing a single value must be written including a comma. **Example:**

```
tup1 = (50,);
```

Tuple indices start at 0, like string indices.

tuple() : Accessing Values in Tuples

To access values in tuple, use the square brackets for access the single element. slicing [start:end] is also available to obtain value available at that index.

Example

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7)
print("tup1[0]: ", tup1[0])          # access to single element
print("tup2[1:5]: ", tup2[1:5])      # access to slice
```

Output:

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

- tuple are immutable, so does NOT contain methods to:
  - eliminate elements
  - insert elements
- 'tuple' object does not support item assignment

  Example:

  ```
  >>>t1=(1,2,3,4,’ciao’,’mondo’,[2,3])
  >>>t1[3]=‘jkjk’
  ```

Output:

```
Traceback (most recent call last):
    File "<pyshell#26>", line 1, in
<module>
    t1[1]=3
    TypeError: 'tuple' object does not
support item assignment
```

tuple() : Delete

Removing individual tuple elements is not possible.
It can be created a new tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the del statement.

Example

```
tup = ('physics', 'chemistry', 1997, 2000)
print tup
del tup
print("After deleting tup : ")
print(tup)
```

This produces an exception raised, because after del tup tuple does not exist any more

Output:

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
   File "test.py", line 9, in <module>
      print tup;
NameError: name 'tup' is not defined
```

tuple() : basic operations

Tuples respond to the + and * operators much like strings;
+ means concatenation
* means repetition
the result is a new tuple, not a string.

Tuples respond to all of the general sequence operations availble on strings:

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

tuple() : Built-in Tuple Functions

Function	Description
cmp(tuple1, tuple2)	Compares elements of both tuples
len(tuple)	Gives the total length of the tuple
max(tuple)	Returns item from the tuple with max value
min(tuple)	Returns item from the tuple with min value
tuple(seq)	Converts a list into tuple

dict{}

Python dictionary is an unordered collection of items.

Elements in a dictionary are key:value pairs.

- values can be of any data type and can repeat,
- keys must be of immutable type (string, number or tuple with immutable elements) and must be unique. Keys are case-sensitive

Each element in a dictionary is identified by the key.
Dictionaries are optimized to retrieve values when the key is known.

Example how to create a dictionary:

```
>>>d={ }                              # empty dictionary
>>>d={1: 'Hello', 'due': 'World'}     # dictionary with two elements
>>>d[1]                               # access to a dictionary element
'hello'
```

dict{} : Creation examples

```
# empty dictionary
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

dict{} : Access elements

In the other container types indexing is used to access values,
Dictionary uses keys to access values.
Key can be used either inside square brackets or with the get() method.

get() returns None if the key is not found.
[ ] returns KeyError if the key is not found.

Example:

```
my_dict = {'name':'Jack', 'age': 26}

print(my_dict['name'])         # Output: Jack

print(my_dict.get('age'))     # Output: 26

# Trying to access keys which doesn't exist throws error (try)
# my_dict.get('address')
# my_dict['address']
```

- keys() and values() functions return respectively the keys and the values present in a dictionary.

dict{} : change or add elements in a dictionary

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.

If the key is already present, value gets updated,
else a new key: value pair is added to the dictionary.

Example:

```
my_dict = {'name':'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

dict{}: delete or remove elements

We can remove a particular item in a dictionary by using the method pop(). This method removes as item with the provided key and returns the value.

The method, popitem() can be used to remove and return an arbitrary item (key, value) form the dictionary.

All the items can be removed at once using the clear() method.

del keyword can be used to remove individual items or the entire dictionary itself.

dict{}: delete or remove elements examples

```
# create a dictionary
squares = {1:1, 2:4, 3:9, 4:16, 5:25}

# remove a particular item
print(squares.pop(4))          # Output: 16
print(squares)                 # Output: {1: 1, 2: 4, 3: 9, 5: 25}

# remove an arbitrary item
print(squares.popitem())       # Output: (1, 1)
print(squares)                 # Output: {2: 4, 3: 9, 5: 25}

# delete a particular item
del squares[5]
print(squares)                 # Output: {2: 4, 3: 9}

# remove all items
squares.clear()
print(squares)                 # Output: {}

# delete the dictionary itself
del squares

# print(squares)               # Throws Error
```

dict{}: built-in functions

Function	Description
all()	Return True if all keys of the dictionary are true (or if the dictionary is empty).
any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries.
sorted()	Return a new sorted list of keys in the dictionary.

Example:

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

print(len(squares))          # Output: 5
print(sorted(squares))       # Output: [1, 3, 5, 7, 9]
```

Exercise: Practice with these functions

dict{}: built-in methods

Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(seq[, v])	Return a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Compares items of two dictionaries.
items()	Return a new sorted list of keys in the dictionary.
keys()	Return a new view of the dictionary's keys.
pop(key[,d])	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
update([other])	Update the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Return a new view of the dictionary's values
has_key(k)	Return True or Falese if key is in the dictionary

dict{}: Built-in methods example

```
marks = {}.fromkeys(['Math','English','Science'], 0)


print(marks)            # Output: {'English': 0, 'Math': 0, 'Science': 0}


for item in marks.items():
    print(item)
list(sorted(marks.keys()))    # Output: ['English', 'Math', 'Science']
```

dict{}: Other operations

Iterating Through a Dictionary
Using a for loop we can iterate though each key in a dictionary.

Example:

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
    print(squares[i])
```

Dictionary Membership Test
We can test if a key is in a dictionary or not using the keyword in. Notice that membership test is for keys only, not for values.

Example:

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

print(1 in squares)             # Output: True
print(2 not in squares)         # Output: True

# membership tests for key only not value
print(49 in squares)            # Output: False
```

dict{}: Python dictionary comprehension

Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair (key: value) followed by for statement inside curly braces {}.

Example to make a dictionary with each item being a pair of a number and its square.

```
squares = {x: x*x for x in range(6)}
print(squares)          # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Equivalent to:

```
odd_squares = {x: x*x for x in range(11) if x%2 == 1}
print(odd_squares)  # Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

A dictionary comprehension can optionally contain more for or if statements.
An optional if statement can filter out items to form the new dictionary.
Example to make dictionary with only odd items.

```
odd_squares = {x: x*x for x in range(11) if x%2 == 1}
print(odd_squares)        # Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

set - fronzenset

Python has two structures to represent sets of elements:

- set is a mutable, unordered collection of etherogeneous objects
- frozenset is an immutable, unordered collection of etherogeneous objects. It is a freezed set

In both cases, elements are unique.

Example:

```
>>>s=set((‘ciao’,1,’Mondo’))
>>>fs=frozenset((‘ciao’,2))
```

- Sets provide methods to modify the data set:
  - insert with add(obj)
  - modify with update(obj)

Example:

```
>>>s=set(('abc','def',1,2,3,'ghi'))
>>>s.add(4)
>>>s.update((‘lmn’,5))
>>>s
set([1, 2, 3, 4, 5, 'abc', 'lmn', 'ghi', 'def'])
```

set - frozenset

- Removal
  - discard(x)
  - remove(x)
  - clear()
  - pop()

Example:

```
>>> s=set([2, 3, 'abc', 'ghi', 'def'])
>>> s
set([2, 3, 'abc', 'def', 'ghi'])
>>> s.remove(3)
>>> s.discard(2)
>>> s.pop()
'abc'
>>> s.clear(); s
set([])
```

set - frozenset

- Both containers contain methods to manage operations :
  - union,
  - intersection,
  - difference,
  - issubset,
  - issuperset

Example:

```
>>>s=set((1,2))
>>>s2=frozenset((2,3,4))
>>>s3=s.union(s2)
>>>s4=s.difference(s2)
>>>s5=s2.intersection(s)
>>>s.issubset(s2)
False
>>>print('s3', s3 , 's4', s4, 's5', s5)
s3
s3 {1, 2, 3, 4} s4 {1} s5 frozenset({2})
```

In both cases data can be of different types.

=> frozenset are immutable, so they can be used to index dictionaries