Chapter 5: Variables and Data Types

Variables

Variables in the Python interpreter are created by assignment and destroyed by the garbage collector, when there are no more references to them.

Variable names must start with a letter or underscore (`_`) and be followed by letters, digits or underscores.

> Uppercase and lowercase letters are considered different. That means **Variables in Python are case sensitive just like C/C++ and Java, so myVar and myvar are different variables**.

Python provides many pre-defined simple data types, such as:

- Numbers (integer, real, complex, boolean ...)
- Strings

Python also provides many complex data types such as collections. The main ones are:

- List
- Tuple
- Dictionary

In Python data type can be either mutable or immutable:

- **Mutable**: allow the contents of the variables to be changed.
- **Immutable**: do not allow the contents of variables to be changed.

In Python, variable names are references that can be changed at execution time.

The most common types and routines are implemented in the form of *builtins*, i.e. they are always available at runtime, without the need to import any library.

Numbers

Python provides following numeric types as *builtins*:

- Integer (*int*): i = 1
- Floating Point real (*float*): f = 3.14

- Complex (*complex*): c = 3 + 4j

In addition to the conventional integers, there are also long integers, whose dimensions are arbitrary and limited by the available memory. Conversions between integer and long are performed automatically. The builtin function `int()` can be used to convert other types to integer, including base changes.

Example:

```
# Converting real to integer
print ('int(3.14) =', int(3.14))
print ('int(3.64) =', int(3.64))

# Converting integer to real
print ('float(5) =', float(5))

# Calculation between integer and real results in real
print ('5.0 / 2 + 3 = ', 5.0 / 2 + 3)

# Integers in other base
print ("int('20', 8) =", int('20', 8)) # base 8
print ("int('20', 16) =", int('20', 16)) # base 16

# Operations with complex numbers
c = 3 + 4j
print ('c =', c)

print ('Real Part:', c.real)
print ('Imaginary Part:', c.imag)
print ('Conjugate:', c.conjugate())
```

```
int(3.14) = 3
int(3.64) = 3
float(5) = 5.0
5.0 / 2 + 3 =  5.5
int('20', 8) = 16
int('20', 16) = 32
c = (3+4j)
Real Part: 3.0
Imaginary Part: 4.0
Conjugate: (3-4j)
```

> NOTE: The real numbers can also be represented in scientific notation, for example: 1.2e22.

Arithmetic Operations:

Python has a number of defined operators for handling numbers through arithmetic calculations, logic operations (that test whether a condition is true or false) or bitwise processing (where the numbers are processed in binary form).

Logical Operations:

- Less than (<)
- Greater than (>)
- Less than or equal to (<=)
- Greater than or equal to (>=)
- Equal to (==)
- Not equal to (!=)

```
def lessThan(x, y):
    if(x < y):
        print("X wins")
    else:
        print("Y wins")

lessThan(22, 4)
```

```
Y wins
```

```
def greaterThan(x, y):
    if(x > y):
        print("X wins")
    else:
        print("Y wins")

greaterThan(2, 4)
```

```
Y wins
```

```
def less_than_or_equal_to(x, y):
    if(x <= y):
        print("X wins")
    else:
        print("Y wins")

less_than_or_equal_to(2, 4)
less_than_or_equal_to(24, 24)
less_than_or_equal_to(24, 4)
```

```
X wins
X wins
Y wins
```

```python
def greater_than_or_equal_to(x, y):
    if(x >= y):
        print("X wins")
    else:
        print("Y wins")

greater_than_or_equal_to(2, 4)
greater_than_or_equal_to(24, 24)
greater_than_or_equal_to(24, 4)
```

```
Y wins
X wins
X wins
```

```python
def equal_to(x, y):
    if(x == y):
        print("X & Y are equal")
    else:
        print("X & Y are different")

equal_to(2, 4)
equal_to(2.2, 2.2)
equal_to(2+1j, 3+1j)
equal_to(3+1j, 3+1j)
```

```
X & Y are different
X & Y are equal
X & Y are different
X & Y are equal
```

```python
def not_equal_to(x, y):
    if(x != y):
        print("X & Y are not equal")
    else:
        print("X & Y are equal")

not_equal_to(2, 4)
not_equal_to(2.2, 2.2)
not_equal_to(2+1j, 3+1j)
not_equal_to(3+1j, 3+1j)
```

```
X & Y are not equal
X & Y are equal
X & Y are not equal
X & Y are equal
```

Bitwise Operations:

- Left Shift (<<)
- Right Shift (>>)
- And (&)
- Or (|)
- Exclusive Or (^)
- Inversion (~)

During the operations, numbers are converted appropriately (eg. `(1.5+4j) + 3` gives `4.5+4j`).

Besides operators, there are also some *builtin* features to handle numeric types: `abs()`, which returns the absolute value of the number, `oct()`, which converts to octal, `hex()`, which converts for hexadecimal, `pow()`, which raises a number by another and `round()`, which returns a real number with the specified rounding.

```
x = 10 -> 1010
y = 11 -> 1011

print("x<<2 = ", x<<2)
print("x =", x)
print("x>>2 = ", x>>2)
print("x&y = ", x&y)
print("x|y = ", x|y)
print("x^y = ", x^y)
print("x =", x)
print("~x = ", ~x)
print("~y = ", ~y)
```

```
x<<2 =  40
x = 10
x>>2 =  2
x&y =  10
x|y =  11
x^y =  1
x = 10
~x =  -11
~y =  -12
```

String

Strings are Python *builtins* datatype for handling text. They are **immutable** thus you can **not add, remove or change** any character in a *string*. To perform these operations, Python needs to create a new *string*.

> > Since Python 3, strings are by default unicode string.

Types:

- Standard String: `s = 'Led Zeppelin'`
- Unicode String: `u = u'Björk'`

> The standard *string* can be converted to *unicode* by using the function `unicode()`.

String can be initialized using:

- With single or double quotes ('', "").
- On several consecutive lines, provided that it's between three single or double quotes (''' ''', """ """).
- Without expansion characters (example: `s = r '\ n'`, where `s` will contain the characters `\` and `n`).

String Operations:

```
s = 'Camel'

# Concatenation
print ('The ' + s + ' ran away!')

# Interpolation
"""
string interpolation (or variable interpolation, variable substitution,
or variable expansion) is the process of evaluating a string literal
containing one or more placeholders, yielding a result in which the
placeholders are replaced with their corresponding values.
"""
print( 'Size of %s => %d' % (s, len(s)))

# String processed as a sequence
for ch in s: print(ch , end='  ') # This

print(".")
print("~"*79)

# Strings are objects
```

```
if s.startswith('C'): print (s.upper())

print(s.lower())
print("~"*79)

# what will happen?
print (3 * s)

# 3 * s is consistent with s + s + s
print(dir(s))
```

```
The Camel ran away!
Size of Camel => 5
C   a   m   e   l   .
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CAMEL
camel
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CamelCamelCamel
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__'
```

```
s = "   Rahman "
age = 10
print(s + str(age))
print(s.strip(), age)
print(s + age)
```

```
   Rahman 10
Rahman 10




---------------------------------------------------------------------------


TypeError                                 Traceback (most recent call last)

<ipython-input-16-d40a63b43035> in <module>()
      3 print(s + str(age))
      4 print(s.strip(), age)
----> 5 print(s + age)


TypeError: must be str, not int
```

```
st = "    Mayank Johri    "
```

```
print(len(st))
s = st.strip()
print(len(s))
print(st.rstrip())
print(st.lstrip())
```

```
20
12
    Mayank Johri
Mayank Johri
```

```
m = "Test String"
x = ["mon", "tues", "wed"]
y = "#"
a = "te sd"
print(y.join(x)) # -> mon, tues, web
print(m.join(y))
print(a.join(y))
print(y.join(a))
```

```
mon#tues#wed
#
#
t#e# #s#d
```

> ??? Resolution: # Since they are single element array this a is not printed, similar to last element printed

The operator `%` is used for string interpolation. The interpolation is more efficient in use of memory than the conventional concatenation.

Symbols used in the interpolation:

- %s: *string*.
- %d: integer.
- %o: octal.
- %x: hexacimal.
- %f: real.
- %e: real exponential.
- %%: percent sign.

Symbols can be used to display numbers in various formats.

Example:

```
# Zeros left
print ('Now is %02d:%02d.' % (16, 30))

# Real (The number after the decimal point specifies how many decimal digits )
print ('Percent: %.1f%%, Exponencial:%.2e' % (5.333, 0.00314))

# Octal and hexadecimal
print ('Decimal: %d, Octal: %o, Hexadecimal: %x' % (10, 10, 10))
```

```
Now is 16:30.
Percent: 5.3%, Exponencial:3.14e-03
Decimal: 10, Octal: 12, Hexadecimal: a
```

format

In addition to interpolation operator `%`, the string method and function `format()` is available.

> The function `format()` can be used only to format one piece of data each time.

Examples:

```
peoples = [('Mayank', 'friend', 'Manish'),
('Mayank', 'reportee', 'Roshan Musheer')]

# Parameters are identified by order
msg = '{0} is {1} of {2}'

for name, function, friend in peoples:
    print(msg.format(name, function, friend))

# Parameters are identified by name
msg = '{greeting}, it is {hour:02d}:{minute:02d}'

print( msg.format(greeting='Good Morning', hour=9, minute=30))
print(msg)
# Builtin function format()
print ('Pi =', format(3.14159, '.3e'))
print ('Pi =', format(3.14159, '.1e'))
```

```
Mayank is friend of Manish
Mayank is reportee of Roshan Musheer
Good Morning, it is 09:30
{greeting}, it is {hour:02d}:{minute:02d}
Pi = 3.142e+00
Pi = 3.1e+00
```

Slices

Slices of *strings* can be obtained by adding indexes between brackets after a *string*.

Slicing strings

Python indexes:

- Start with zero.
- Count from the end if they are negative.
- Can be defined as sections, in the form `[start: end + 1: step]`. If not set the start, it will be considered as zero. If not set end + 1, it will be considered the size of the object. The step (between characters), if not set, is 1.

> **!!! TIP !!!**: It is possible to invert *strings* by using a negative step:

```
p = "Mayank Johri"
print(p[0:2])
print (p[::-1])
print(p[:-3])
print("implemented"[::-2]) # ->
print("implemented"[-5::-2])
print("implemented"[:-7:-2])
print("implemented"[::2])
print("implemented"[1::2])
print("implemented"[1:4:2])
```

```
Ma
irhoJ knayaM
Mayank Jo
dteepi
eepi
dte
ipeetd
mlmne
ml
```

> ?1 -> starts from i(**m**) and ends

`str` in-build module

Strings implement all of the common sequence operations, along with the additional methods described below.

```
myStr = "maya deploy, version: 0.0.3 "

print(myStr.capitalize())
print(myStr.center(60))
print(myStr.center(60, "*"))
print(myStr.center(10, "*"))

print(myStr.count('a'))
print(myStr.count('e'))

print(myStr.endswith('all'))
print(myStr.endswith('.0.3 '))

print(myStr.find("g"))
print(myStr.find("e"))
```

```
Maya deploy, version: 0.0.3
                maya deploy, version: 0.0.3
****************maya deploy, version: 0.0.3 ****************
maya deploy, version: 0.0.3
2
2
False
True
-1
6
```

> **Note**: The find() method should be used only if you need to know the position of sub. To check if sub is a substring or not, use the in operator:

```
print("g" in myStr)
```

```
True
```

```
c = "one"
print(c.isalpha())
c = "1"
print(c.isalpha())
```

```
True
False
```

```
```

```
superscripts = "\u00B2"
five = "\u0A6B"
#str.isdecimal() (Only Decimal Numbers)
print(five)
print(c.isdecimal())
print(five.isdecimal())
print("10 ->", "10".isdecimal())
print("10.001".isdecimal())

str = u"this 2009";
print(str.isdecimal())

str = u"23443434";
print(str.isdecimal())
print(fractions.isdecimal())
```

```
੫
True
True
10 -> True
False
False
True
False
```

```
# str.isdigit() (Decimals, Subscripts, Superscripts)
fractions = "\u00BC"
print(fractions)
print(c.isdigit())
print(fractions.isdigit())
print(five.isdigit())

print("10".isdigit())
str = u"this 2009";
print(str.isdigit())

str = u"23443.434";
print(str.isdigit())
```

```
¼
True
False
True
True
False
```

```
False
```

```python
print(superscripts)
print(superscripts.isdigit())
print(superscripts.isdecimal())
print(superscripts+superscripts)
print(fractions+fractions)
```

```
²
True
False
²²
¼¼
```

```python
# str.isnumeric() (Digits, Fractions, Subscripts, Superscripts, Roman Numerals, Curren
print(fractions)
print(fractions.isnumeric())
print(five.isnumeric())
```

```
¼
True
True
```

```python
print(myStr.isalnum())
print("one".isalnum())
print("thirteen".isalnum())
```

```
False
True
True
```

String Module

Various functions for dealing with text are implemented in the module *string*.

```python
import string
```

```
# the alphabet
print(dir(string))
a = string.ascii_letters
print(a)
# Shifting left the alphabet
b = a[1:] + a[0]
print(b)
```

```
['Formatter', 'Template', '_ChainMap', '_TemplateMetaclass', '__all__', '__builtins__'
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
bcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZa
```

Template

The module also implements a type called *Template*, which is a model *string* that can be filled through a dictionary. Identifiers are initialized by a dollar sign ($) and may be surrounded by curly braces, to avoid confusion.

Example:

```
import string

# Creates a template string
st = string.Template('$warning occurred in $when $$what')

# Fills the model with a dictionary
s = st.substitute({'warning': 'Lack of electricity',
    'when': 'April 3, 2002'})

# Shows:
# Lack of electricity occurred in April 3, 2002
print(s)
```

```
Lack of electricity occurred in April 3, 2002 $what
```

```
# Unicode String
u = u'Hüsker Dü'
# Convert to str
s = u.encode('latin1')
print (s, '=>', type(s))

# String str
s = 'Hüsker Dü'
```

```
# u = s.decode('latin1')

print (repr(u), '=>', type(u))
```

```
b'H\xfcsker D\xfc' => <class 'bytes'>
'Hüsker Dü' => <class 'str'>
```

To use both methods, it is necessary to pass as an argument the compliant coding. The most used are "latin1" "utf8".

Lists

Lists are collections of heterogeneous objects, which can be of any type, including other lists.

Lists in the Python are mutable and can be changed at any time. Lists can be sliced in the same way as *strings*, but as the lists are mutable, it is possible to make assignments to the list items.

Syntax:

```
list = [a, b, ..., z]
```

Common operations with lists:

```
fruits = ['Apple', 'Mango', 'Grapes', 'Jackfruit', 'Apple', 'Banana', 'Grapes', [1, "C

# # processing the entire list
# for prog in fruits:
#      print(prog,  end=", ")

# #
# print("")


# # Including
# fruits.append('Camel')
# print(fruits)




# # # # Removing

## Removing the second instance of Grapes
x = 0
y = 0
```

```
for fruit in fruits:
    if x == 1 and fruit == 'Grapes':
#           del (fruits[y])
        fruits.pop(y)
    elif fruit == 'Grapes':
        x = 1
    y +=1

print(fruits)


# # # fruits.remove('Grapes')
# # print(fruits)
# # fruits.append("Grapes")

# while('Grapes' in fruits ):
#     fruits.remove('Grapes')
#     print(fruits)
# # # # Ordering
# # These will work on only homogeneous list and will fail for heterogeneous
# fruits.sort()
# print(fruits)

# # # # Inverting
# fruits.reverse()
# print(fruits)

# # # # prints with number order
# for i, prog in enumerate(fruits):
#     print( i + 1, '=>', prog)
# print(len(fruits))
# # # prints from de second item
# print (fruits[1:])
```

```
['Apple', 'Mango', 'Grapes', 'Jackfruit', 'Apple', 'Banana', [1, 'Orange']]
```

The function `enumerate()` returns a tuple of two elements in each iteration: a sequence number and an item from the corresponding sequence.

The list has a `pop()` method that helps the implementation of queues and stacks:

```
my_list = ['A', 'B', 'C']
print ('list:', my_list)

# # The empty list is evaluated as false
# while my_list:
#     # In queues, the first item is the first to go out
#     # pop(0) removes and returns the first item
#     print ('Left', my_list.pop(0), ', remain', len(my_list))

# # More items on the list
# my_list += ['D', 'E', 'F']
# print ('list:', my_list)

while my_list:
    # On stacks, the first item is the last to go out
    # pop() removes and retorns the last item
    print ('Left', my_list.pop(), ', remain', len(my_list))
```

```
list: ['A', 'B', 'C']
Left C , remain 2
Left B , remain 1
Left A , remain 0
```

The sort (*sort*) and reversal (*reverse*) operations are performed in the list and do not create new lists.

Tuples

Similar to lists, but immutable: it's not possible to append, delete or make assignments to the items.

Syntax:

```
my_tuple = (a, b, ..., z)
```

The parentheses are optional.

Feature: a tuple with only one element is represented as:

t1 = (1,)

The tuple elements can be referenced the same way as the elements of a list:

```
first_element = tuple[0]
```

Lists can be converted into tuples:

```
my_tuple = tuple(my_list)
```

And tuples can be converted into lists:

```
my_list = list(my_tuple)
```

While tuple can contain mutable elements, these elements can not undergo assignment, as this would change the reference to the object.

Example (using the interactive mode):

```
t = ([1, 2], 4)
print(t)
print(" :: Error :: ")

# t[0] = 3
# print(t)

# t[0] = [1, 2, 3]
# print(t)

# t[0].append(3)
# print(t)
t[0][0] = [1, 2, 3]
print(t)

ta = (1, 2, 3, 4, 5)

for a in ta:
    print (a)

ta1 = [1, 2, 3, 4, 5]
for a in ta1:
    print(a)
```

```
([1, 2], 4)
 :: Error ::
([[1, 2, 3], 2], 4)
1
2
3
4
5
1
2
3
4
5
```

NOTE: Tuples are more efficient than conventional lists, as they consume less computing resources (memory) because they are simpler structures the same way *immutable* strings are in relation to *mutable* strings.

Lists Versus Tuples

Tuples are used to collect an immutable ordered list of elements. This means that to a tuple (**limitation**):

- elements can't be added, thus There’s no append() or extend() method for tuples,
- elements can't be removed, thus Tuples have no remove() or pop() method,

So, if we have a constant set of values and only we will iterate through it than use a tuple instead of a list as It is faster & safer than working with lists, as the tuples contain “write-protect” data.

Other types of sequences

Also in the *builtins*, Python provides:

- *set*: mutable sequence univocal (without repetitions) unordered.
- *frozenset*: immutable sequence univocal unordered.

Both types implement set operations, such as: union, intersection e difference.

Example:

```
test = (1, 2, 3, "Aasdf", [1, 3, 5])
print(test)
test[-1][1]= "test"
print(test)
test[-1].append("testing")
print(test)
test[-1] = ""
print(test)
```

```
(1, 2, 3, 'Aasdf', [1, 3, 5])
(1, 2, 3, 'Aasdf', [1, 'test', 5])
(1, 2, 3, 'Aasdf', [1, 'test', 5, 'testing'])



---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-37-e7c27041ba5e> in <module>()
      5 test[-1].append("testing")
      6 print(test)
----> 7 test[-1] = ""
      8 print(test)
```

```
TypeError: 'tuple' object does not support item assignment
```

```
s1 = (1,2,3)
# print(s1)
# print(len(s1))
# print(s1.append(4))
s2 = ([1,2], 4)
print(s2)

# s2[0][1] = [5]
x = s2[0]
x[1] = [5]
x.append(5)
print(type(x))
s2[0].append(5)
# s2[0] = [6]
print(s2)
```

```
([1, 2], 4)
<class 'list'>
([1, [5], 5, 5], 4)
```

```
s = ((1,2),(21,22),(31,31),(41,42))
print(s)
print(s[0][1])
print(s[1][1])
print(s[1][0])
print(s[0])
```

```
((1, 2), (21, 22), (31, 31), (41, 42))
2
22
21
(1, 2)
```

```
# Data sets
s1 = set(range(3))
s2 = set(range(10, 7, -1))
s3 = set(range(2, 10, 2))
s4 = [8, 9]

# Shows the data
print ('s1:', s1, '\ns2:', s2, '\ns3:', s3)
```

```python
# Union
s1s2 = s1.union(s2)
print ('Union of s1 and s2:', s1s2)
s2s1 = s2.union(s1)
print ('Union of s2 and s1:', s2s1)

# Difference
print ('Difference with s3:', s1s2.difference(s3))

# Intersectiono
print ('Intersection with s3:', s1s2.intersection(s3))

# Tests if a set includes the other
if s1.issuperset([1, 2]):
    print ('s1 includes 1 and 2')

# Tests if there is no common elements
if s1.isdisjoint(s2):
    print ('s1 and s2 have no common elements')

# Tests if a set includes the other
if s2.issuperset(s4):
    print ('s2 includes all items from s4')
else:
    print("s2 does not include all items from s4")

# Tests if there is no common elements
if s2.isdisjoint(s3):
    print ('s2 and s3 have no common elements')
else:
    print("s2 and s3 have common elements")
```

```
s1: {0, 1, 2}
s2: {8, 9, 10}
s3: {8, 2, 4, 6}
Union of s1 and s2: {0, 1, 2, 8, 9, 10}
Union of s2 and s1: {0, 1, 2, 8, 9, 10}
Difference with s3: {0, 1, 10, 9}
Intersection with s3: {8, 2}
s1 includes 1 and 2
s1 and s2 have no common elements
s2 includes all items from s4
s2 and s3 have common elements
```

When one list is converted to a *set*, the repetitions are discarded.

In version 2.6, a *builtin* type for mutable characters list, called *bytearray* is also available.

Dictionaries

A dictionary is a list of associations composed by a unique key and corresponding structures. Dictionaries are mutable, like lists.

The key must be an immutable type, usually strings, but can also be tuples or numeric types. On the other hand the items of dictionaries can be either mutable or immutable. The Python dictionary provides no guarantee that the keys are ordered.

Syntax:

```
dictionary = {'a': a, 'b': b, ..., 'z': z}
```

Structure:

Structure of a dictionary

Example of a dictionary:

```
dic = {'name': 'Dabar', 'band': 'Honey'}
```

Acessing elements:

```
print dic['name']
```

Adding elements:

```
dic['value'] = '120'
```

Removing one elemento from a dictionary:

```
del dic['value']
```

Getting the items, keys and values:

```
items = dic.items()
keys = dic.keys()
values = dic.values()
```

Examples with dictionaries:

```
dic = {'name': 'Dabar', 'name': 'Dabar New', 'band': 'Honey'}
print(dic)
```

```
{'name': 'Dabar New', 'band': 'Honey'}
```

```
# Progs and their albums
progs = {'Yes': ['Close To The Edge', 'Fragile'],
    'Genesis': ['Foxtrot', 'The Nursery Crime'],
    'ELP': ['Brain Salad Surgery']}

# More progs
progs['King Crimson'] = ['Red', 'Discipline']

# items() returns a list of
# tuples with key and value
for albums in progs.items():
    print(albums)

# for prog, albums in progs.items():
#     print(prog, '=>', albums)


# for prog, albums in progs.values():
#     print(prog, '=>', albums)

for prog in progs:
    print(prog, "=>", progs[prog])

# If there is 'ELP', removes
if 'ELP' in progs:
    del progs['ELP']

print(progs)
```

```
('Yes', ['Close To The Edge', 'Fragile'])
('Genesis', ['Foxtrot', 'The Nursery Crime'])
('ELP', ['Brain Salad Surgery'])
('King Crimson', ['Red', 'Discipline'])
Yes => ['Close To The Edge', 'Fragile']
Genesis => ['Foxtrot', 'The Nursery Crime']
ELP => ['Brain Salad Surgery']
King Crimson => ['Red', 'Discipline']
{'Yes': ['Close To The Edge', 'Fragile'], 'Genesis': ['Foxtrot', 'The Nursery Crime'],
```

```
# Matrix in form of string
```

```python
matrix = '''0 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 4 0 0
0 0 0 0 0 0 0 3 0 0 0 0
0 0 0 0 0 0 5 0 0 0 0 0
0 0 0 0 6 0 0 0 0 0 0 0'''

mat = {}

# split the matrix in lines
for row, line in enumerate(matrix.splitlines()):

    # Splits the line int cols
    for col, column in enumerate(line.split()):

        column = int(column)
        # Places the column in the result,
        # if it is differente from zero
        if column:
            mat[row, col] = column

print (mat)
# The counting starts with zero
print ('Complete matrix size:', (row + 1) * (col + 1))
print ('Sparse matrix size:', len(mat))
```

```
{(5, 4): 6, (3, 7): 3, (1, 0): 9, (4, 6): 5, (2, 9): 4}
Complete matrix size: 72
Sparse matrix size: 5
```

The sparse matrix is a good solution for processing structures in which most of the items remain empty, like spreadsheets for example.

True, False and Null

In Python, the boolean type (*bool*) is a specialization of the integer type (*int*). The *True* value is equal to 1, while the *False* value is equal to zero.

The following values are considered false:

- `False`.
- `None` (null).
- `0` (zero).
- `''` (empty string).
- `[]` (empty list).
- `()` (empty tuple).

- `{}` (emtpy dicionary).
- Other structures with size equal zero.

All other objects out of that list are considered true.

The object *None*, which is of type *NoneType*, in Python represents the null and is evaluated as false by the interpreter.

Boolean Operators

With logical operators it is possible to build more complex conditions to control conditional jumps and loops.

Boolean operators in Python are: *and*, *or* , *not* , *is* , *in*.

- `and` : returns a true value if and only if it receives two expressions that are true.
- `or` : returns a false value if and only if it receives two expressions that are false.
- `not` : returns false if it receives a true expression and vice versa.
- `is` : returns true if it receives two references to the same object false otherwise.
- `in` : returns true if you receive an item and a list and the item occur one or more times in the list false otherwise.

The calculation of the resulting operation *and* is as follows: if the first expression is true, the result will be the second expression, otherwise it will be the first.

As for the operator *or* if the first expression is false, the result will be the second expression, otherwise it will be the first. For other operators, the return will be of type bool (True or False).

Examples:

```
print (0 and 3) # Shows 0
print (2 and 3 )# Shows 3

print (0 or 3) # Shows 3
print (2 or 3) # Shows 2

print (not 0) # Shows True
print (not 2) # Shows False
print (2 in (2, 3)) # Shows True
print (2 is 3) # Shows False
```

```
0
3
3
2
```

```
True
False
True
False
```

Besides boolean operators, there are the functions `all()`, which returns true when all of the items in the sequence passed as parameters are true, and `any()`, which returns true if any item is true.

Excercise

1. What will be the output of the following code snippets?

a.

```
a=[1,2,3,4,5,6,7,8,9]
print(a[::2])
```

b.

```
a=[1,2,3,4,5,6,7,8,9]
a[::2]=10,20,30,40,50,60
print(a)
```

c.

```
a=[1,2,3,4,5]
print(a[3:0:-1])
```

d.

```
arr = [[1, 2, 3, 4],
       [4, 5, 6, 7],
       [8, 9, 10, 11],
       [12, 13, 14, 15]]
for i in range(0, 4):
    print(arr[i].pop())
```

e.

```
arr = [1, 2, 3, 4, 5, 6]
for i in range(1, 6):
    arr[i - 1] = arr[i]
for i in range(0, 6):
    print(arr[i], end = " ")
```

f.

```
nums = set([1,1,2,3,3,3,4])
print len(nums)
```

g.

```
numbers = [1, 2, 3, 4]

numbers.append([5,6,7,8])

print len(numbers)
```

h.

```
names1 = ['Amir', 'Barry', 'Chales', 'Dao']
names2 = names1
names3 = names1[:]

names2[0] = 'Alice'
names3[1] = 'Bob'

sum = 0
for ls in (names1, names2, names3):
    if ls[0] == 'Alice':
        sum += 1
    if ls[1] == 'Bob':
        sum += 10

print(sum)
```

i.

```
names1 = ['Amir', 'Barry', 'Chales', 'Dao']

loc = names1.index("Edward")

print (loc)
```

j.

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
```

```
print len(list1 + list2)
```

k.

```
list1 = [1, 2, 3, 8, 4]
list2 = [5, 6, 7, 8, 2]

print len(set(list1 + list2))
```

1. Write a Python script to add key to a dictionary.
   > Sample Dictionary : {0: 10, 1: 20} Expected Result : {0: 10, 1: 20, 2: 30}

2. Write a Python script to concatenate following dictionaries to create a new one.
   > Sample Dictionary : dic1={1:10, 2:20} dic2={3:30, 4:40} dic3={5:50,6:60} Expected Result : {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}

3. Write a Python script to check if a given key already exists in a dictionary.
4. Write a Python program to iterate over dictionaries using for loops.

5. Write a Python script to generate and print a dictionary that contains number (between 1 and n) in the form (x, x*x). Sample Dictionary (n = 5) : Expected Output : {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

6. Write a Python script to merge two Python dictionaries.

7. Write a Python script to sort (ascending and descending) a dictionary by value.
8. Write a Python program to sum all the items in a dictionary.
9. Write a Python program to multiply all the items in a dictionary.
10. Write a Python program to remove a key from a dictionary.
11. Write a Python program to map two lists into a dictionary.
12. Write a Python program to sort a dictionary by key.
13. Write a Python program to get the maximum and minimum value in a dictionary.
14. **TODO: Write a Python program to get a dictionary from an object's fields.**
15. Write a Python program to remove duplicates (in terms of value) from Dictionary.
16. Write a Python program to check a dictionary is empty or not.