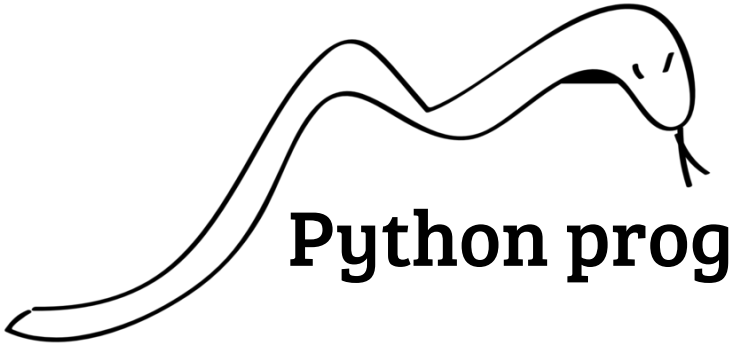


Python programming language

Academy

Slajdy dodatkowe - 1/3

Przypominamy o zabraniu ze sobą sprzętu do kodowania!

W ramach kolejnych (2) zajęć będziemy wspólnie przygotowywać program w oparciu o materiał na nich omawiany. Zalecamy zabranie:

- Laptopa
- Ładowarki
- ***Python 3.6.x***
- Dowolne IDE do pracy na nim lub notatnik

Slajdy dodatkowe - 2/3

W ramach powtórki zajęć pytanie:

1. Jaka jest różnica pomiędzy funkcjami **range()** oraz **enumerate()** omawianymi na zajęciach?
2. Co zwróci polecenie:

a) >>> range(1,3,1)
b) >>> enumerate ([10,20,30])

Slajdy dodatkowe - 3/3

- Skąd czerpać wiedzę o nowo poznanych funkcjach w pythonie?

- Można korzystać z forów (stack overflow), youtube i facebooka. Ale zdecydowanie polecamy zajrzeć do **>>>dokumentacji<<<**.
- Dokumentacja w samym pythonie i wielu modułach z nim współpracujących jest przejrzyście opisana i zawiera wiele przykładów które ułatwiają zrozumienie jak działa konkretna funkcja.
- Przykład dla enumerate(): https://docs.python.org/3/library/functions.html#enumerate

About authors

Imię

Nazwisko

Numer Albumu

Kierunek/ Wydział

IMPORTANT

AĄBCĆDEĘFGHIJKLŁMNŃ
OÓPQRSŚTUVWXYZŹŻ
aąbcćdeęfghijklłmnńoópq
rsśtuvwxyzźż 1234567890

POGROMCY BAZGROŁÓW

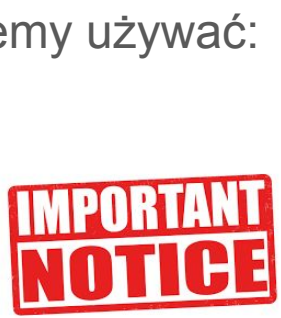
Rozkład jazdy na akademię:

1. Podstawy i obsługa języka Python. ** **CRASHCOURSE** **
2. Operowanie na plikach. (pyxl)
3. Python for science. Czyli trochę o modułach Numpy, Scipy i Matplotlib.
4. Jak sprawić by komputer zrozumiał naszą mowę? (speechrecognition, pyaudio)
5. Aplikacje okienkowe. (wxpython, sklearn)
6. W pełni funkcjonalne aplikacje internetowe. (django, sqlite3)
7. Podstawy tworzenia gier czyli jak wąż ma zjeść tyle jabłek? (pygame)
8. *Cython most do połączenia języka Python z C/C++*

Python
3.6.6

Na wszystkich zajęciach będziemy używać:

Python 3.6.6



Interpreter znajdziemy na stronie:
https://www.python.org/downloads/

Dla dociekliwych => więcej o różnicach:
https://wiki.python.org/moin/Python2orPython3

Basics

Python is a programming,scripting, dynamic, strongly typed language with simple, and yet effective approach to **OOP**. The “Tab” button is of a most common use.

Python was created in the early 90’s by **Guido van Rossum** at Stichting Mathematisch Centrum (CWI) as a successor of a language called ABC. For next 10 years, Guido was working on python with various institutions, eventually creating BeOpen PythonLabs team, which moved to Digital Creations Corporation (now it’s different name). In 2001, Python Software Foundation (PSF) was created there.

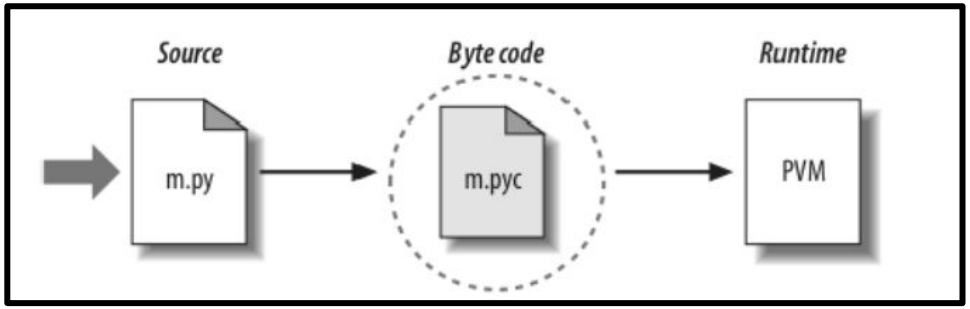
scripting vs programming: https://goo.gl/QFquZv

dynamic vs static language: https://goo.gl/U8hC1n

Implementations & compilation

CPython is the original and most-maintained python implementation, written in C. Most of built-in modules in python **are written in C** (for time-effectiveness reasons). Other python implementations are: Jython (Java), IronPython (C#), or even PyPy (python implementation written in python).

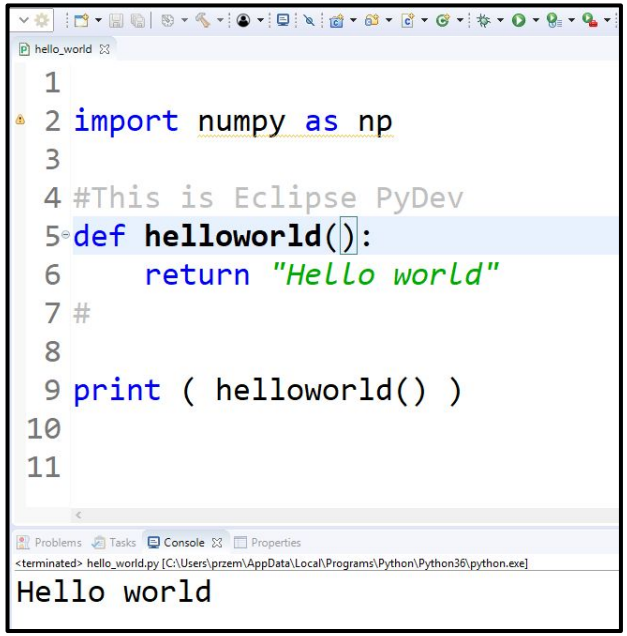
Compilation steps:



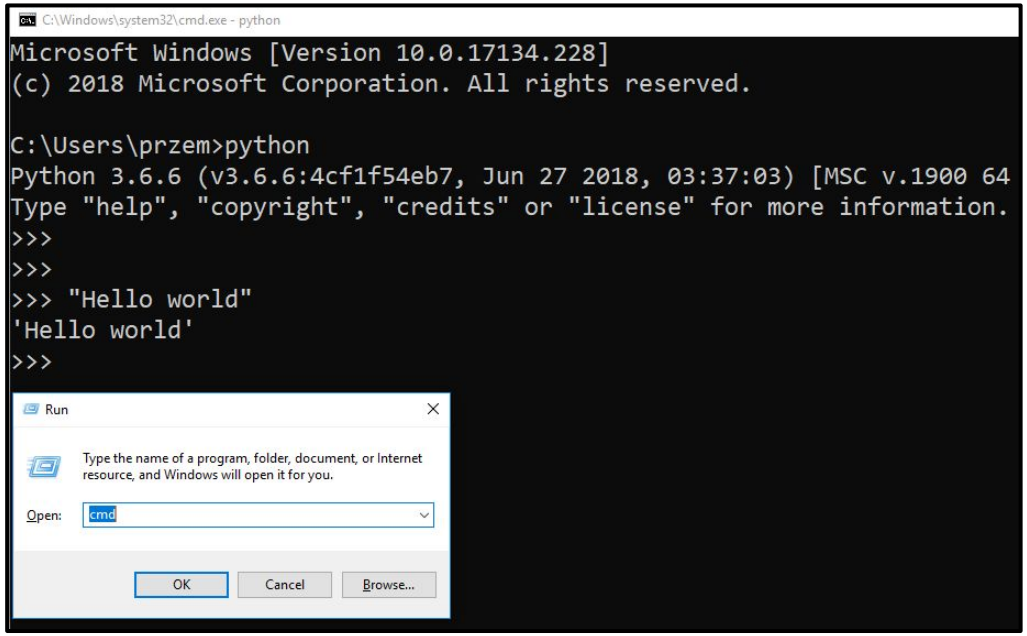
CPython vs Python: https://goo.gl/fph4yZ Python and JIT: https://goo.gl/NnsdPY

Writing modes & most known IDE's

script mode



interpreted mode



Writing modes & most known IDE's

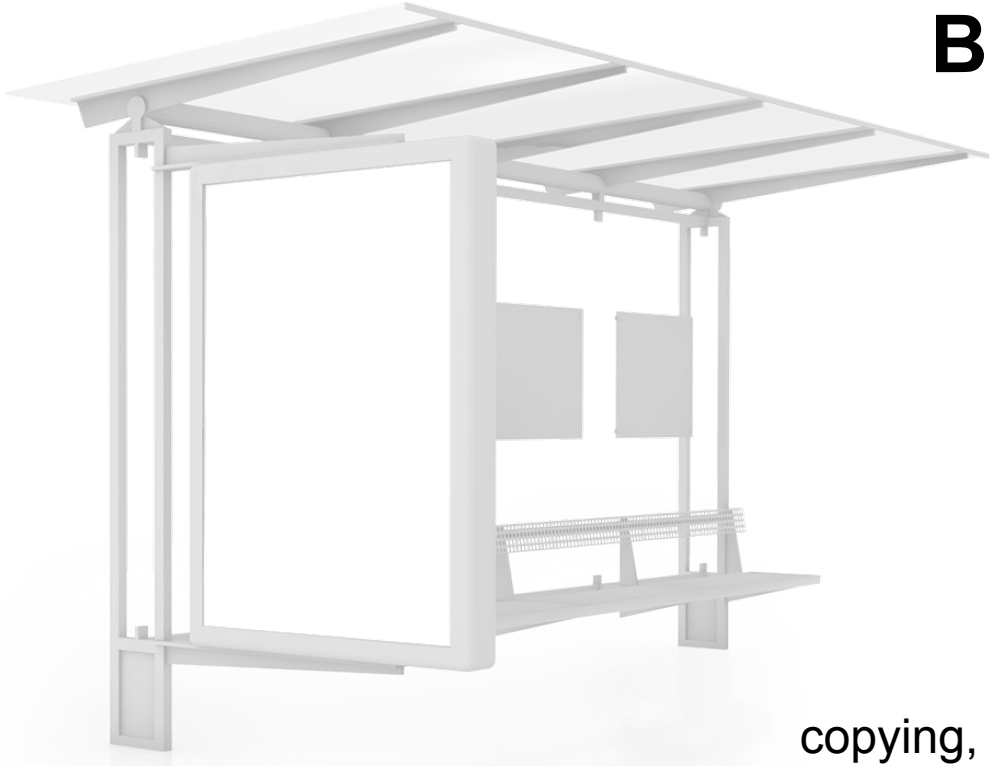
PyCharm

PyDev

IDLE

Spyder

Atom



Bus Stations for today

Chapter I

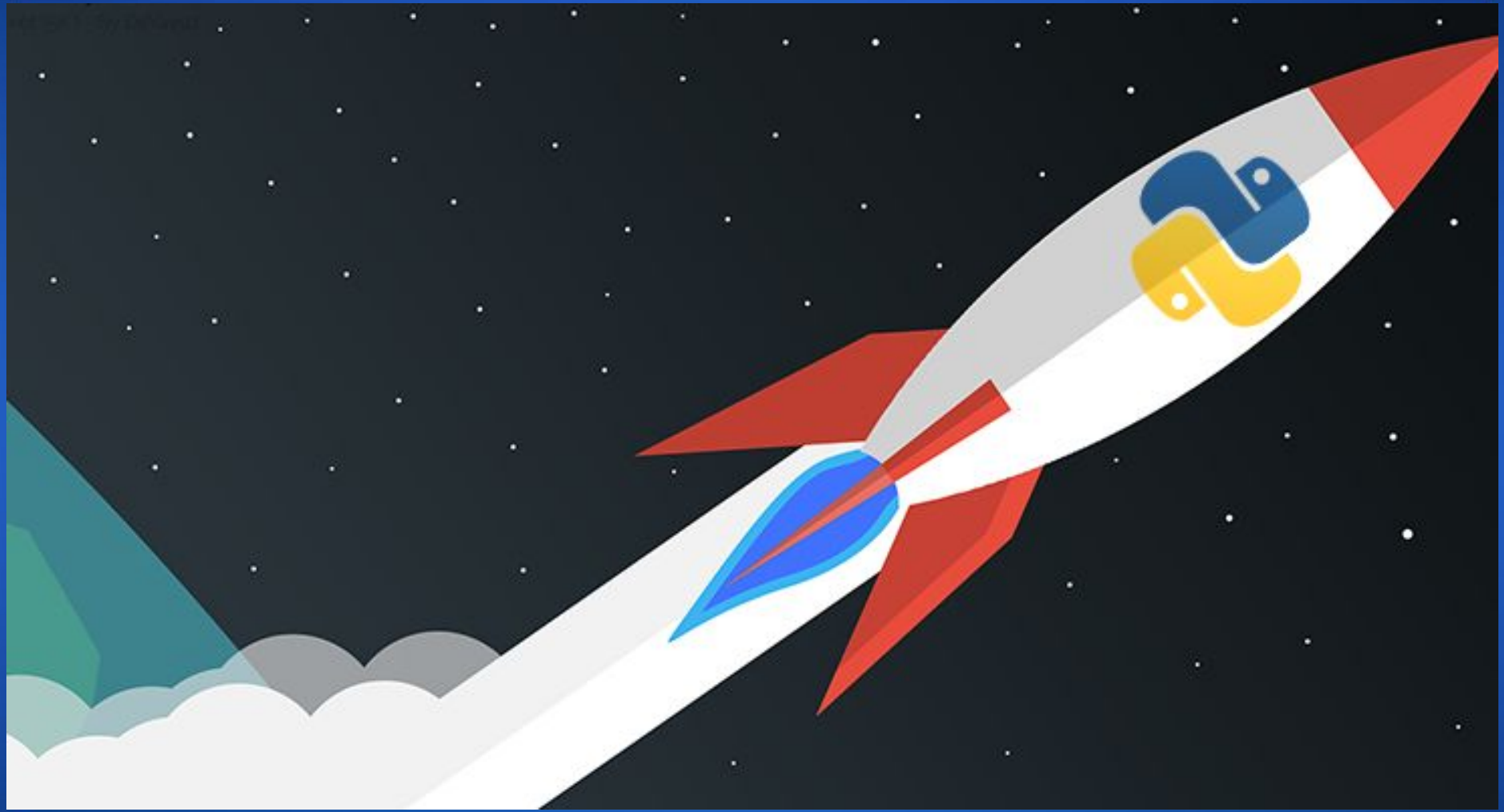
data types & operations

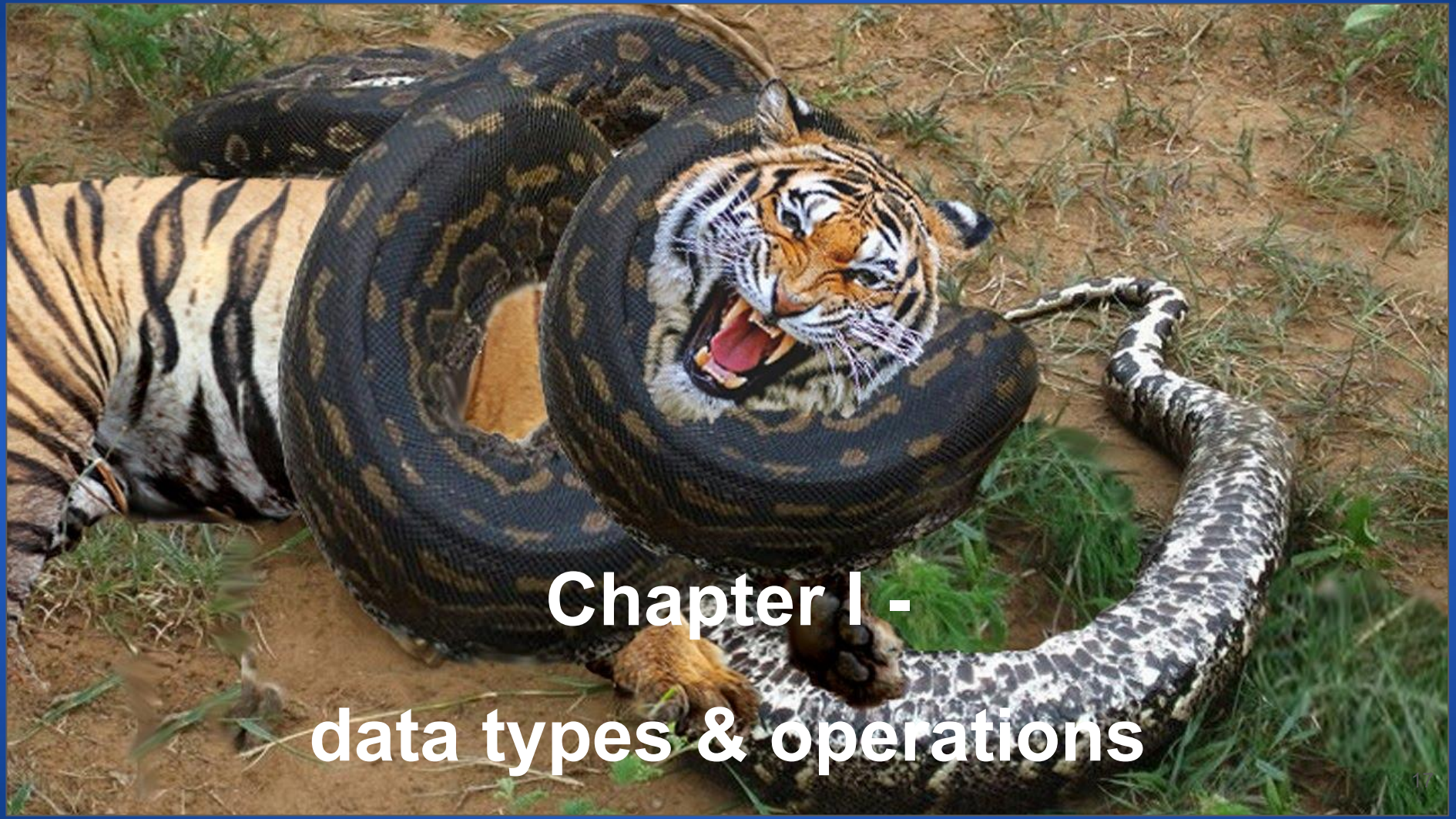
Chapter II

indexing, comprehension, functions

Chapter III

copying, modules, dunder methods and more



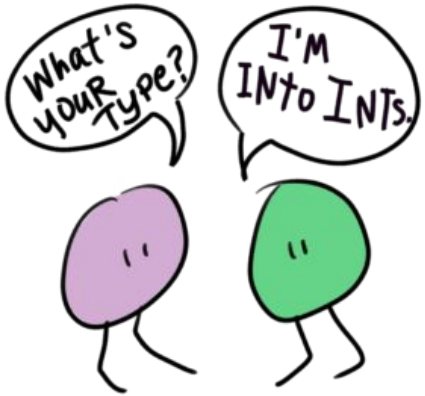

Chapter I -
data types & operations

Data types

Python variables does not have explicit type. That means names are just connected to an object, which resides somewhere in the memory, but what is inside that object, is not of a particular interest to a compiler.

However, depending on how the object was constructed, python is able to inform programmer what its type is - using function **type()**.

There are 6 elementary data types in python:



Data types & basic operations

number string list tuple dictionary set

```
>>>
>>> type (1)
<class 'int'>
>>>
```

```
>>>
>>> type (1.0)
<class 'float'>
>>>
```

```
>>>
>>> type ( 1.85+2j )
<class 'complex'>
>>>
```

Data types & basic operations

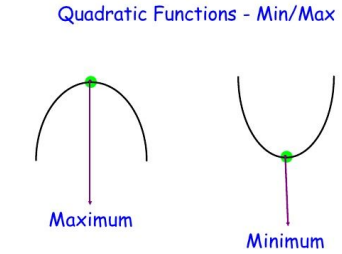
number string list tuple dictionary set

```
>>> 4 + 9
13
>>> 2.3 / 5
0.45999999999999996
>>> 2.3 // 5
0.0
>>>
```

```
>>> (1.5 + 2j) * (2.5 - 1.3j)
(6.35+3.05j)
>>>
>>> 2.5 + 3
5.5
>>> max(1,3)
3
```

$a + bi$

Real part Imaginary part



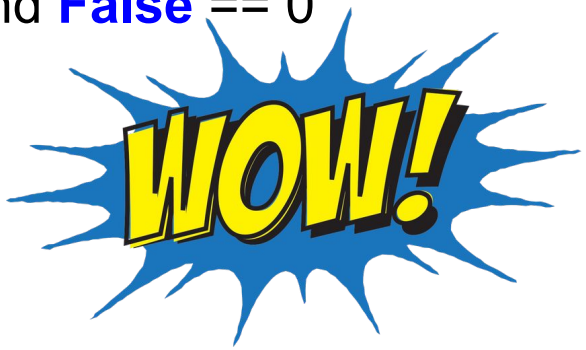

SO, I JUST DIVIDE BY ZERO AND THEN..

Data types & basic operations

number string list tuple dictionary set

```
>>> a,b = 1,2
>>> a
1
>>> a,b = b,a
>>> a
2
```

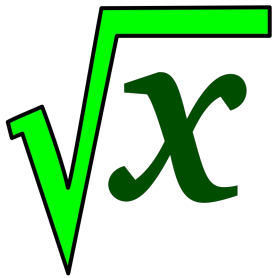
```
>>> True and False == 0
True
>>> 2 ** 3
8
>>> 2 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```



Data types & basic operations

number string list tuple dictionary set

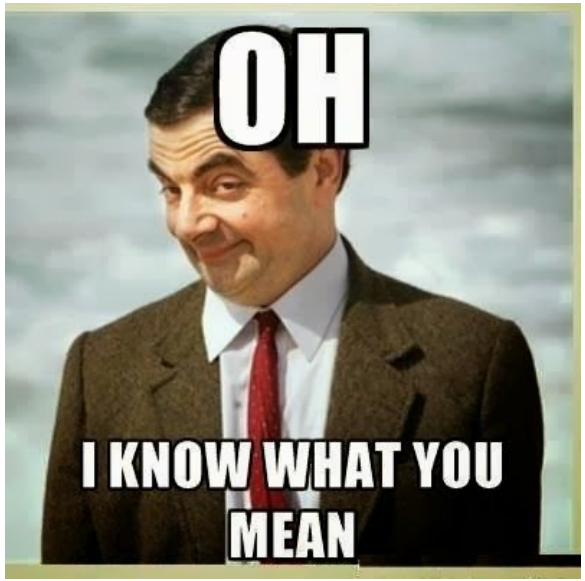
```
>>> from cmath import sqrt
>>> sqrt(-2)
1.4142135623730951j
>>>
```




```
>>> from math import\
... sqrt,pi
>>> sqrt(-pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

Data types & basic operations

number **string** list tuple dictionary set



Data types & basic operations

number **string** list tuple dictionary set

```
>>> type ("This is my string")
<class 'str'>
```

```
>>> type ( 's' )
<class 'str'>
```


```
>>> type ( '' )
<class 'str'>
```

Data types & basic operations

number **string** list tuple dictionary set

```
>>> istr = input("Wassup?: ")
Wassup?: ain't bad, yo
>>> "Busta Rhymes? {}".format(istr)
"Busta Rhymes? ain't bad, yo"
>>>
```

```
>>> 'a' * 3
'aaa'
>>> 'abc'.find('b')
1
>>>
```

Data types & basic operations

number **string** list tuple dictionary set

```
>>> s1 = 'string is ' + 'IMMUTABLE'
>>> s1
'string is IMMUTABLE'
>>> s1[3] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Data types & basic operations

number **string** list tuple dictionary set

```
>>> s1 = 'This is a string'
>>> s1.split('-')
['This is a string']
>>> s2 = s1.split()
>>> s2
['This', 'is', 'a', 'string']
>>> '-'.join(s2)
'This-is-a-string'
```

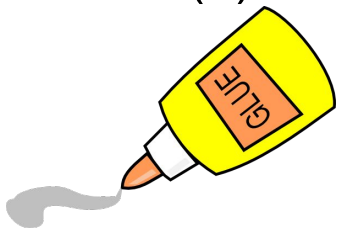
```
>>> a = 'a'
>>> b = 'b'
>>> 'a' 'b'
'ab'
>>> a b
  File "<stdin>", line 1
    a b
      ^
SyntaxError: invalid syntax
```

Data types & basic operations

number **string** list tuple dictionary set

```
>>> 'small string'.capitalize()
'Small string'
>>> 'small string'.upper()
'SMALL STRING'
>>> import string
>>> string.digits
'0123456789'
```


```
>>> s1 = 'Six: '+str(6)
>>> s1
'Six: 6'
>>>
```

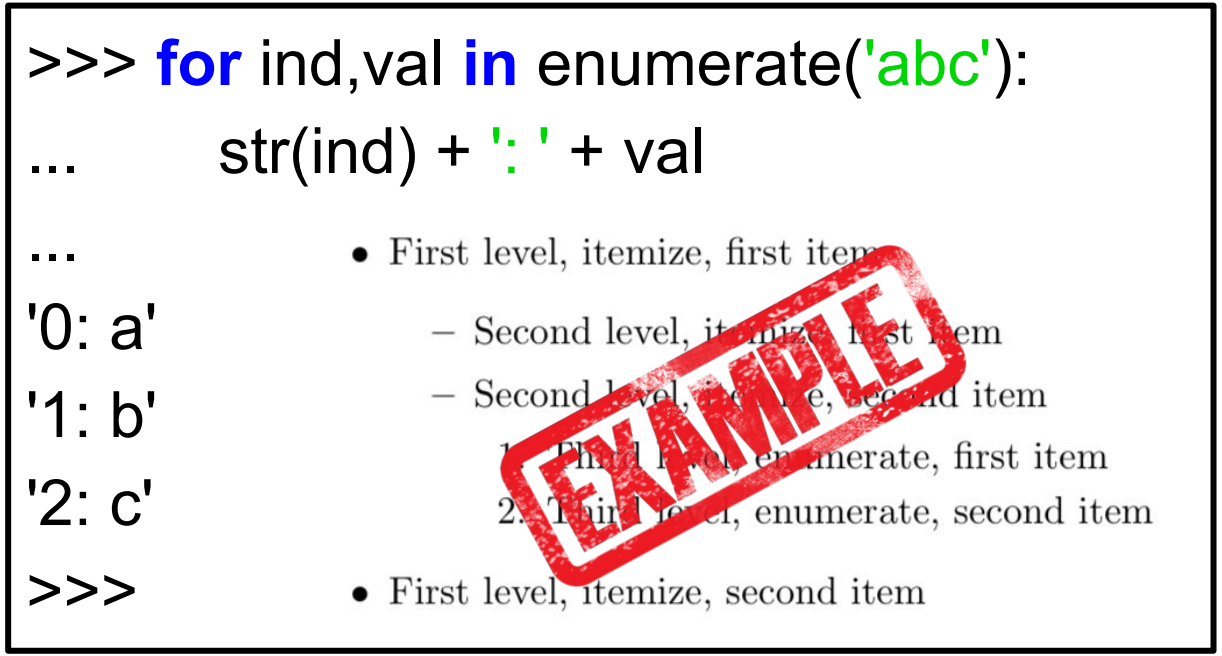



```
>>> 'a?c'.replace('?','b')
'abc'
>>>
```

Data types & basic operations

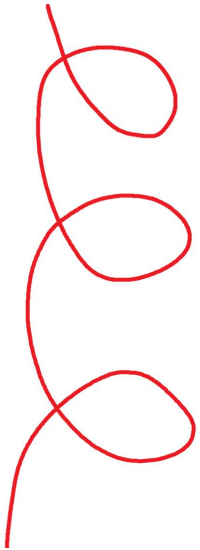
number **string** list tuple dictionary set

```
>>> for ind,val in enumerate('abc'):
...         str(ind) + ': ' + val
...
'0: a'
'1: b'
'2: c'
>>>
```

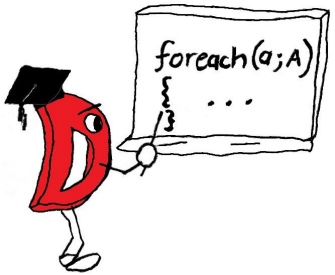


Data types & basic operations

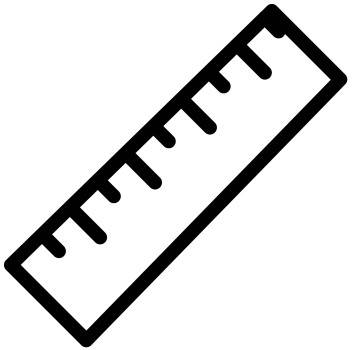
number **string** list tuple dictionary set



```
>>> str = ''
>>> for i in 'abc':
...     str += i
...
>>> str
'abc'
>>>
```




```
>>> len('abc')
3
>>> s1 = 'abc'
>>> s1[0:2]
'ab'
>>> 'abc'[0:2]
'ab'
```



Data types & basic operations

number **string** list tuple dictionary set

```
>>> for i in range('abc'):
...     print(i)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object cannot be interpreted
as an integer
```

Data types & basic operations

number string **list** tuple dictionary set

```
>>> L1 = [0,1,2]
>>> type(L1)
<class 'list'>
```

```
>>> L4 = [1;2;3]
  File "<stdin>", line 1
    L4 = [1;2;3]
          ^
SyntaxError: invalid syntax
>>>
```

Data types & basic operations

number string **list** tuple dictionary set

```
>>> L2 = list('a','b','c')
>>> type(L2)
<class 'list'>
```

```
>>> L3 = [1,'a',[1,2]]
>>> type(L3)
<class 'list'>
```

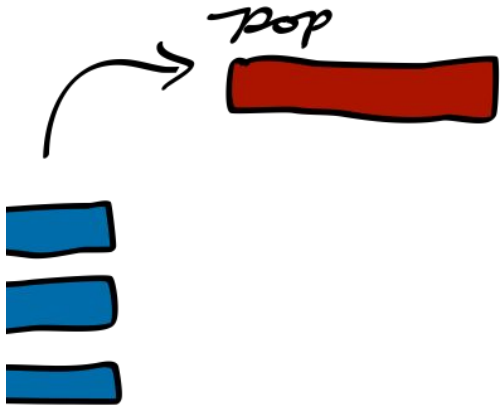
```
>>> letters = ['a','b','c']
>>> letters.index('c')
2
```

Data types & basic operations

number string **list** tuple dictionary set

```
>>> l1 = [0,[1,2],3]
>>> l1[1]
[1, 2]
>>> l1[1] = 1
>>> l1
[0, 1, 3]
```

```
>>> l1 = [0,1,2,3]
>>> l1.pop()
3
>>> l1
[0, 1, 2]
>>> l1.pop(1)
1
```



Data types & basic operations

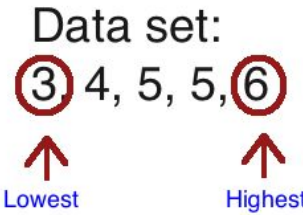
number string **list** tuple dictionary set

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
```

```
>>> l1 = list(range(3))
>>> l1
[0, 1, 2]
```

```
>>> l1 = range(0,6)
>>> l1
range(0, 6)
>>> type(l1)
<class 'range'>
>>>
```

The range is the difference between the lowest value and the highest value.



Data types & basic operations

number string **list** tuple dictionary set

```
>>> mtx = [[0,1,2],
...        [3,4,5]]
>>> mtx[1][1] == 4
True
>>> mtx[1,1] == 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not tuple
```

Data types & basic operations

number string **list** tuple dictionary set

```
>>> help(slice)
Help on class slice in module builtins:

class slice(object)
 |  slice(stop)
 |  slice(start, stop[, step])
 |
 |  Create a slice object.  This is used for extended slicing
...
```

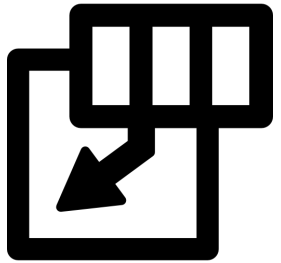
Data types & basic operations

number string **list** tuple dictionary set

```
>>> slice(2)
slice(None, 2, None)

>>> mtx[slice(2)]
[[0, 1, 2], [3, 4, 5]]
```

```
>>> l1 = [1,4,3]
>>> index = 1
>>> value = 2
>>> l1.insert(index,value)
>>>l1.sort()
>>>l1
[1,2,3,4]
```



Data types & basic operations

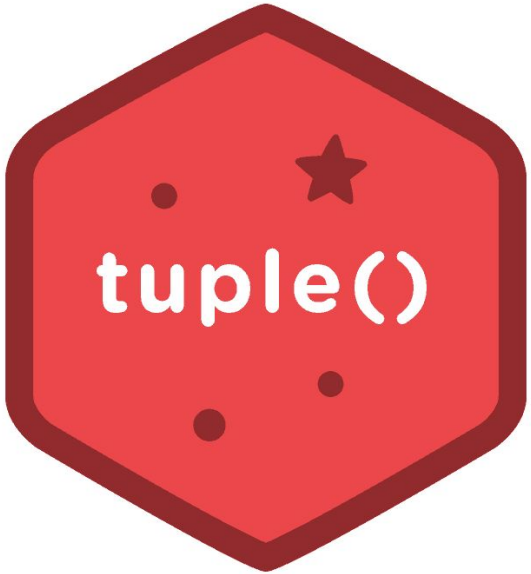
number string list **tuple** dictionary set

```
>>> t1 = (1,2)
>>> type(t1)
<class 'tuple'>
```

```
>>> t2 = 1,2
>>> type(t2)
<class 'tuple'>
```

```
>>> type ((0,1,[2,3],(4)))
<class 'tuple'>
```

```
>>> t3 = 1,2 + 3,4
>>> 1,2 + 3,4
(1, 5, 4)
>>> (1,2) + (3,4)
(1, 2, 3, 4)
```



Data types & basic operations

number string list **tuple** dictionary set

```
>>> t1 = 1,2,'c'
>>> t1[1] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t1.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'pop'
```

Data types & basic operations

number string list tuple **dictionary** set

```
>>> type ( {} )
<class 'dict'>
>>>
>>> dt = {'key':'value'}
>>> type(dt)
<class 'dict'>
```

```
>>> dt1 = {1:1,'a':5,3.0:'c'}
>>> dt1[3.0]
'c'
>>> dt1['a'] = 0
>>> dt1
{1: 1, 'a': 0, 3.0: 'c'}
```

Data types & basic operations

number string list tuple **dictionary** set

```
>>> z = {1:'a', 2: 'b'}
>>> z[2]
'b'
>>> z['b']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'
```

Look-up

Data types & basic operations

number string list tuple **dictionary** set

```
>>> def rlookup(mdict, val):
...     for i in mdict:
...         if mdict[i] == val:
...             return i
...
>>> z
{1: 'a', 2: 'b'}
>>> rlookup(z,'a')
1
```

Reverse look-up

Data types & basic operations

number string list tuple **dictionary** set

```
>>> d1 = {1:'a', 2:'b', 3:'c'}
>>> 'a' in d1
False
>>> 'a' in d1.values()
True
>>>
```

Data types & basic operations

number string list tuple dictionary **set**

```
>>> s1 = set()
>>>
>>> type(s1)
<class 'set'>
```

```
>>> s2 = set([1,1,2,3])
>>> s2.add(1)
>>> s2
{1, 2, 3}
```

```
>>> s3 = {1,2,'a',(1,2)}
>>> s3
{(1, 2), 1, 2, 'a'}
```

Data types & basic operations

number string list tuple dictionary **set**

```
>>> s4 = set([1,2,2,3,3])
>>> s4.add(4)
>>> s4.update([3,4,5])
>>> s4
{1, 2, 3, 4, 5}
>>> s4.add({1,2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Data types & basic operations

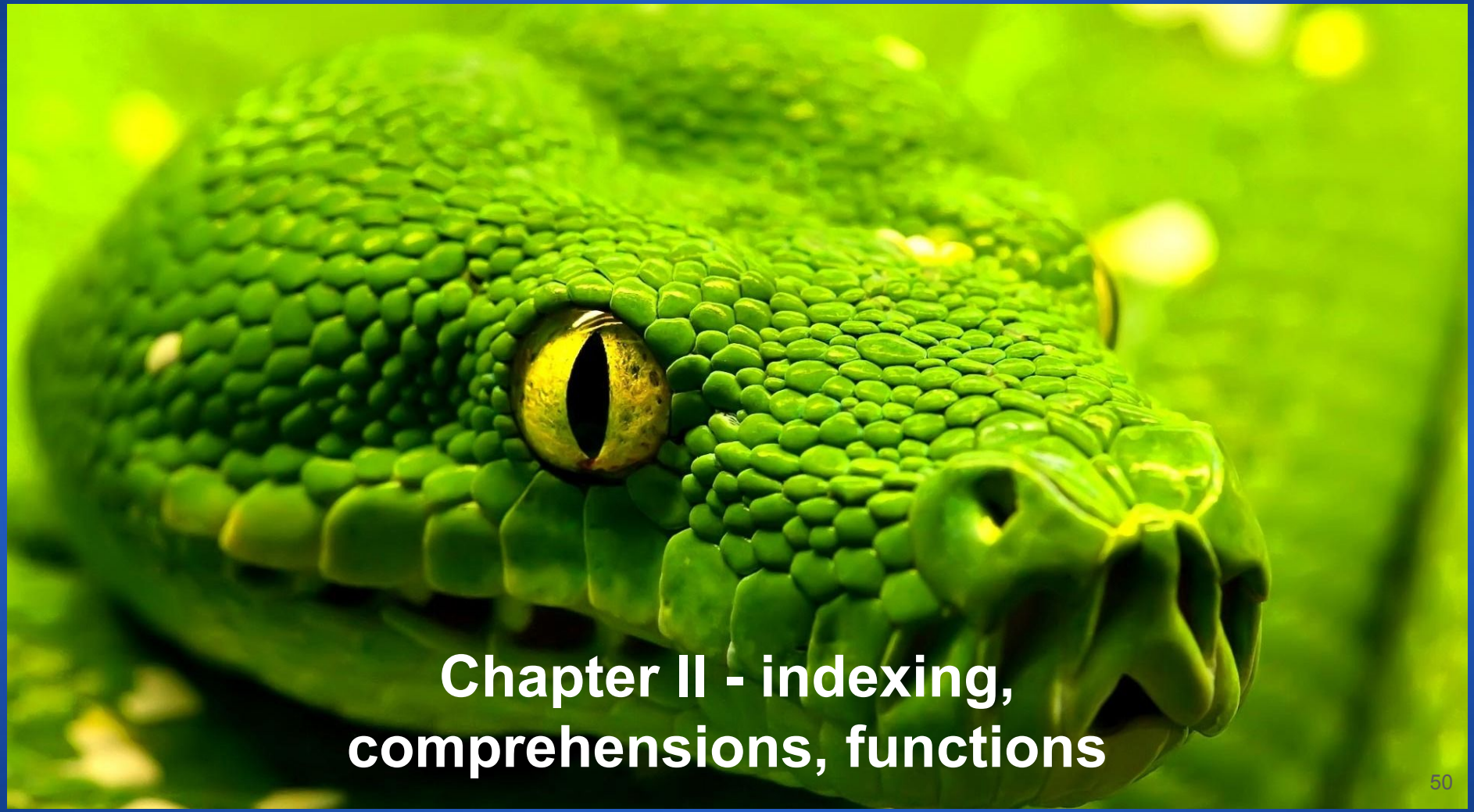
number string list tuple dictionary **set**

```
>>> s4.add([1,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> s4.add((1,2))
>>> s4
{(1, 2), 1, 2, 3, 4, 5}
```

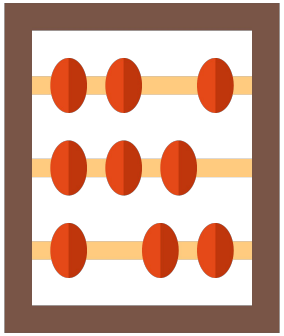
Data types & basic operations

number string list tuple dictionary **set**

```
>>> s5 = {1,2,3}
>>> s5.discard(2)
>>> s5
{1, 3}
>>> s5[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support
indexing
```


Chapter II - indexing,
comprehensions, functions

Indexing



Basic indexing
iterables in python:

iterable[start:end]

- You can omit zeros
- Indexing forward starts from
  0 … size – 1
- Indexing backward starts from
  -size … -1

```
>>> mlist = list( range(10) )
>>> mlist[0:3]
[0, 1, 2]
>>> mlist[:3]
[0, 1, 2]
>>> mlist[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> mlist[-3]
7
>>> mlist[:3] + mlist[-3:]
[0, 1, 2, 7, 8, 9]
```

Indexing

Indexing with steps:

iterable[start:end:step]

- Reverse iterables easily
- Find even & odd numbers
- Iterate through iterables using iterators (more info soon)

```
>>> mlist = list(range(10))
>>> mlist[::2]
[0, 2, 4, 6, 8]
>>> mlist[:6:3]
[0, 3]
>>> mlist[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

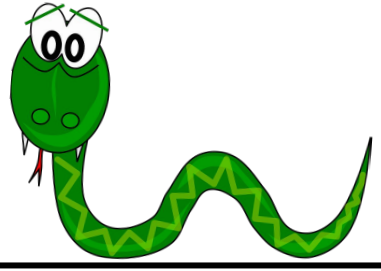
SUMMARY

[6:10]

0	1	2	3	4	5	6	7	8	9	10	11
M	o	n	t	y		P	y	t	h	o	n
–12	–11	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

[–12:–7]

Indexing



Experiments

```
>>> mlist[1,7,8] += 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not tuple
>>> mlist[ mlist > 2 ] += 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'list' and 'int'
```

more: https://goo.gl/ug94Xh

List comprehensions

traditional way:

```
>>> l1 =[]
>>> for i in range(3):
...     l1.append(i)
...
>>> l1
[0, 1, 2]
```

list comprehensions:

```
>>> l2 = [i for i in range(3)]
>>> l2
[0, 1, 2]
>>> l3 = [i ** 3 for i in range(3)]
>>> l3
[0, 1, 8]
```

List comprehensions

The rule: [**expression** **loop** **condition**]

```
>>> l4 = [i for i in range(10) if i%2 ==0]
>>> l4
[0, 2, 4, 6, 8]
>>>
>>> l5 = [i**2 for i in range(10) if i%3]
>>> l5
[1, 4, 16, 25, 49, 64]
>>>
```

If, else, for, while statements

```
>>> import random
>>> rand_var = random.randint(0,4)
>>> if rand_var > 2:
...     True
... elif rand_var < 2:
...     False
…else:
        None
```

```
>>> 'you can' if 0 == 1 else 'you cant'
'you cant'
>>>
```

If, else, for, while statements

```
>>> x = 3
>>> while x > 0:
...       x
...       x -= 1
...
3
2
1
```

```
>>> enumerate('abc')
<enumerate object at 0x0000016A24684B40>
>>> l1 = list( enumerate('abc') )
>>> for x,y in l1:
...     x,y
...
(0, 'a')
(1, 'b')
(2, 'c')
```

Functions – return, yield

```
1  def fun():
2      print(‘This is a fun’)
3      return 1,2
4
5  fun()
6
```

```
7  This is a fun
8  (1,2)
9
```

```
1  def fun():
2      print('This is a fun')
3      return 1,2
4
5  myfun = fun()
6  print('-------')
7  print(myfun)
```

```
8  This is a fun
9  -------
1  (1,2)
```

Functions – return, yield

```
1  def fun():
2      def fun2():
3          print('This is a fun')
4      return fun2
5
6  print( fun() )
7  myvar = fun()
8  print( myvar() )
```

```
9  <function fun.<locals>.fun2 at 0x000026412639620>
10 this is fun2
```

Functions – return, yield

```
1
2
3
4
5
6
7
8
9
10
```

```
def fun():
        yield 1
        yield 2


my_var = fun()
print(my_var)
```

```
<generator object fun at 0x0000026412639620>
```

Functions – default arguments

```
1  def fun(a=1, b=2, c=3):
2      print(a,b,c)
3
4  #calling:
5  fun()
6  fun(4,6,5)
7  fun(a=4,c=6,b=5)
8
```

```
9  1 2 3
10 4 6 5
11 4 5 6
```

Functions – lambda



```
1  def f1 (x) : return x ** 2
2  def f2 (x) : return x ** 3
3  def f3 (x) : return x ** 4
4
5  l = [f1,f2,f3]
6
7  print("Functions: ",end='')
8  for i in l:
9      print(i(4), ' ',end='')
10
11 Functions: 16 64 256
12
```

```
1  l1 = [lambda x: x ** 2,
2        lambda x: x ** 3,
3        lambda x: x ** 4]
4
5  print("Lambda: ",end='')
6  for i in l1:
7      print(i(4), ' ',end='')
8
9
10
11 Lambda: 16 64 256
12
```



Functions – lambda

```
1  def fun(x):
2      return x + 3
3
4  print(fun(3))
5  lam = lambda x: x+3
6  print(lam(3))
7
8
9
10 6
11 6
12
```

```
1  def fun():
2      ac = lambda n: n ** 3
3      return ac
4
5  my_lam = fun()
6  print ( my_lam(3) )
7
8
9
10
11 27
12
```

Functions – pass

```
1
2
3
4
5
6
7
8
9
10
```

```
def fun():
    pass

print(fun())
```

None

Functions – variables scope

```
1
2
3
4
5
6
7
8
9
10
11
12
```

```
my_var = 1
def fun():
#printing my_var here
#gives unbound local error
        my_var = 3
        print(“Inside: ”,my_var)

fun()
print(my_var1)
```

```
None
```

Functions – passing objects

```
1
2
3
4
5
6
7
8
9
10
11
12
```

```
# * for lists, tuples
def fun(a, *b):
        print (a,b)

v1 = 'a'
l1 = [1,2,3]

fun(v1,l1)
```

```
a ([1,2,3],)
```

Functions – unpacking variables

```
1   # ** for dicts
2   def fun (a, *b, **c):
3         print ( a,b,c )
4   #also *args, **kwargs
5   v1 = ‘a’
6   l1  = [1,2,3]
7   d1 = {1: ‘a’, 2:’b’}
8
9   fun( v1, l1, d1)
10  fun( v1, l1, x=1, y=1 )
11  a ([1, 2, 3], {1: 'a', 2: 'b'}) {}
12  a ([1, 2, 3],) {'x': 1, 'y': 2}
```

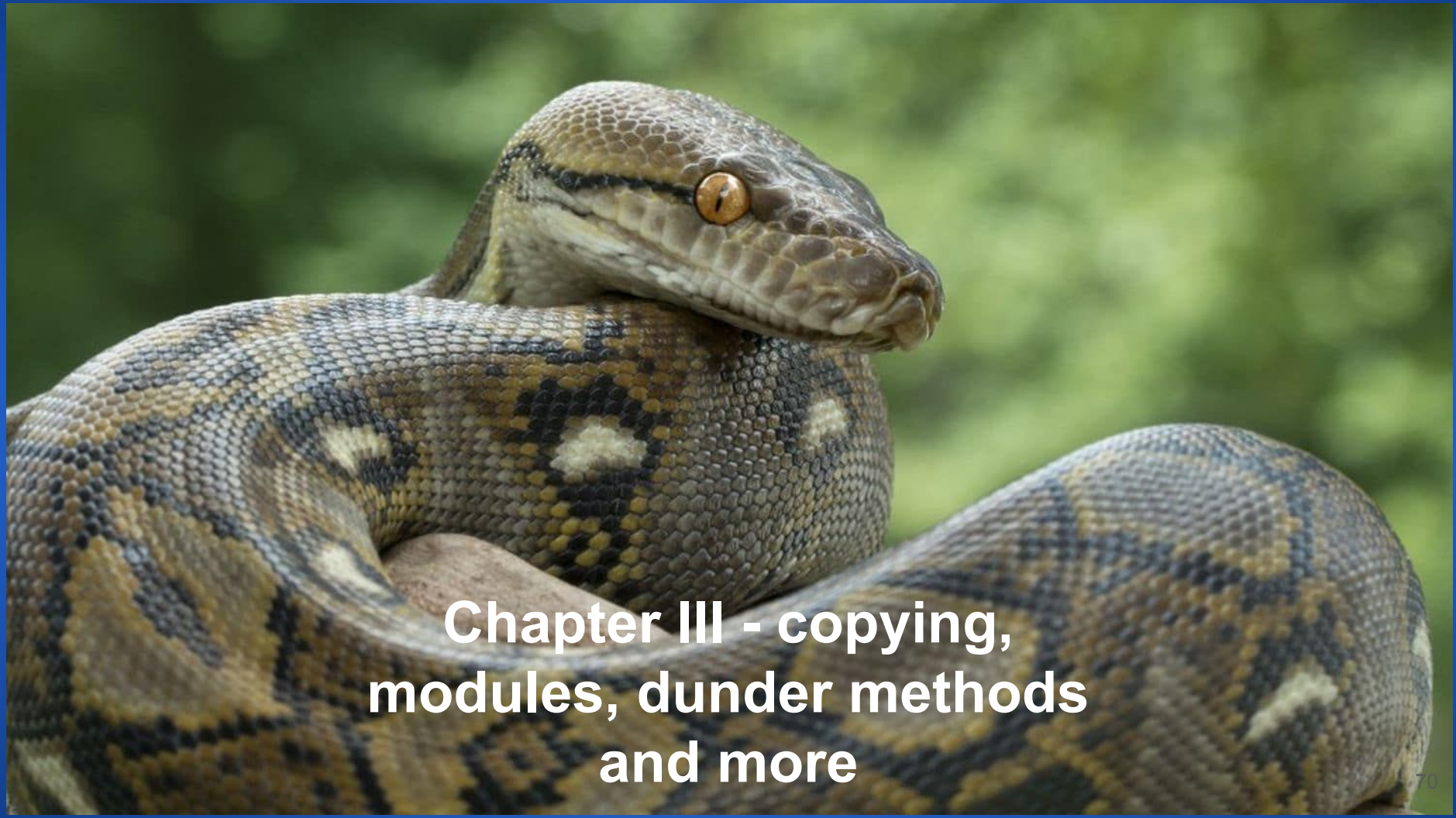
understanding asterisk in py:
https://goo.gl/99SCFH

https://goo.gl/ABTXp4

Functions – unpacking variables

```
1  def fun(a,b):
2      print (a,b)
3
4  l1 = (1,2)
5  d1 = {'a':1,'b':2}
6  fun (*l1)
7  fun (**d1)
```

```
8  1 2
9  1 2
```


Chapter III - copying,
modules, dunder methods
and more

shallow VS deep copy

reference

```
>>> a = [1,2]
>>> b = a
>>> b.pop()
2
>>> a
[1]
```

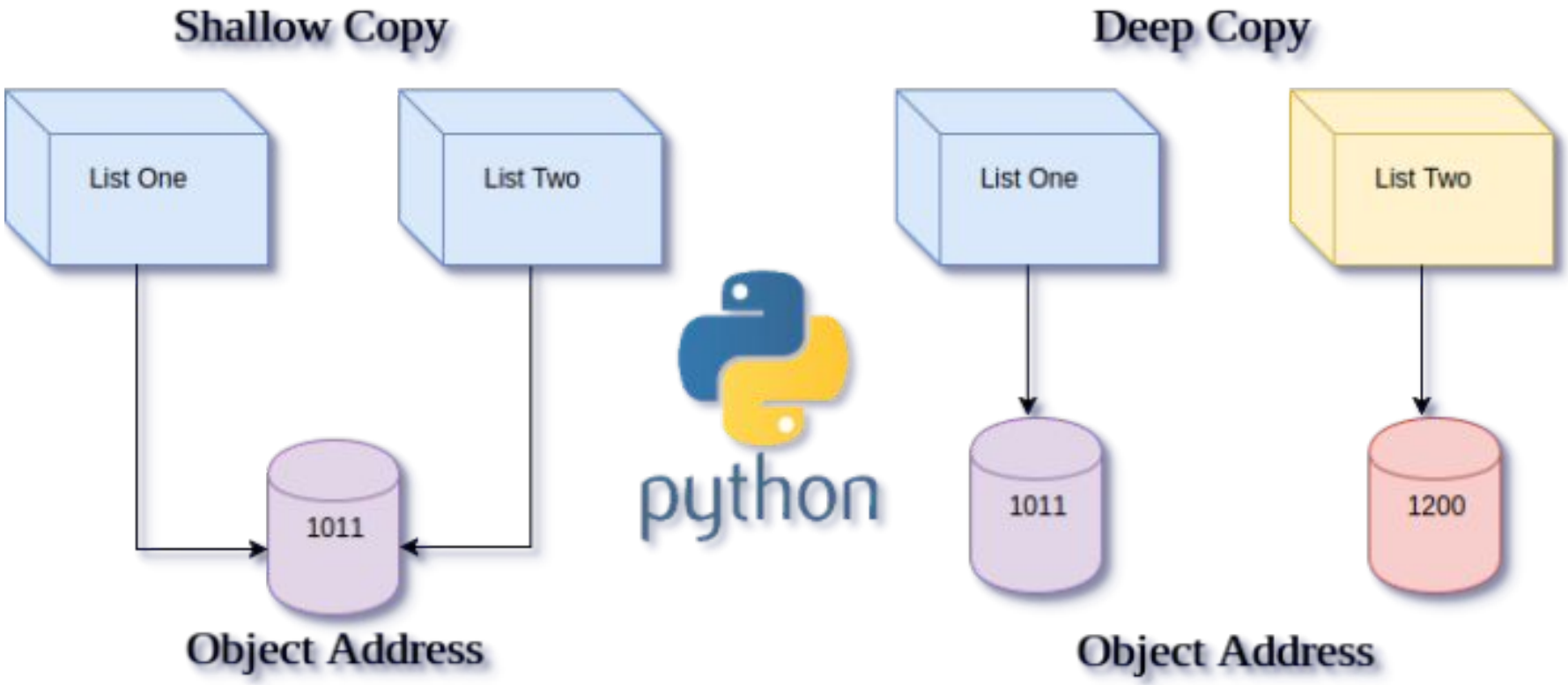
shallow copy

```
>>> a = [1,2]
>>> b = a.copy()
>>>
>>> b.pop()
2
>>> a
[1, 2]
```

deep copy

```
>>> from copy import deepcopy
>>> a = [1,2]
>>> b = a.copy()
>>> b.pop()
2
>>> a
[1, 2]
```

So what is the difference between shallow and deep copy?


Shallow Copy
List One
List Two
1011
Object Address
python
Deep Copy
List One
List Two
1011
1200
Object Address

shallow VS deep copy

Passing argument to a function:

by reference

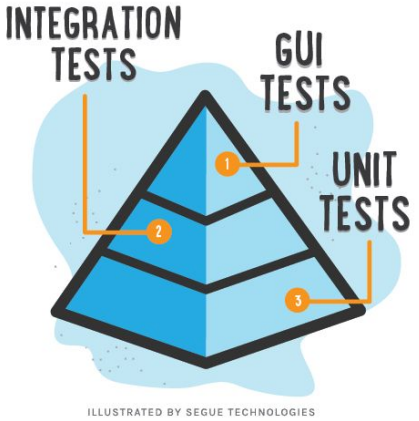
```
1  def fun(a):
2      a[0] = 100
3
4  my1 = [1,2]
5  fun(my1)
6
7  print (my1)
8
9  [100, 2]
```

by copy

```
1  def fun(a):
2      a[0] = 100
3
4  my1 = [1,2]
5  fun ( my1.copy() )
6
7  print( my1 )
8
9  [1, 2]
```

Unit testing


```
1   def mul (a,b):
2       """
3       type some comment here …
4       >>> pow(2,3)
5       8
6       >>> pow(3,4)
7       81
8       """
9       return a ** b
10  if __name__ == "__main__":
11      import doctest
12      doctest.testmod(verbose=True,
13      optionflags=doctest.NORMALIZE_WHITESPACE)
14      2 tests in __main__.pow
15  2 tests in 2 items.
16  2 passed and 0 failed.
17  Test passed.
```



Exceptions

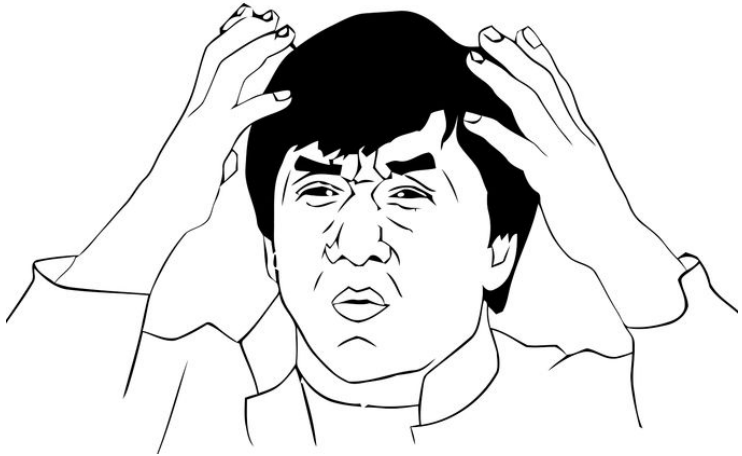
```
1   res = 0
2   try:
3       res = 2 / res
4   except Exception as ex:
5       print(”err: ”,+str(ex))
6   else:
7       print (”Will execute if no error occurred”)
8   finally:
9       print(”Will execute either way”)
```

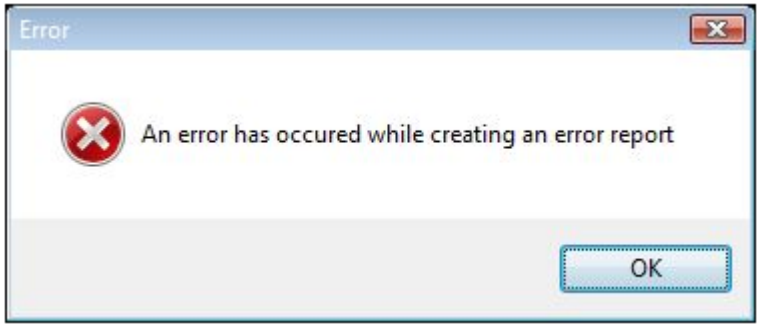
EXCEPTION
PANIC!!!!!

```
10  err: Division by zero
11  Will execute either way
```

Exceptions – raising own error

```
1   b = 0
2   try:
3       if b == 0:
4           raise ValueError('WTF?!')
5
6       print( b / 1 )
7   except Exception as ex:
8       print (str (ex) )
9
10
11  WTF?!
```





Modules & packages

my_module1

```
1  x = 10
2
3
4
5
6
7
```

my_module2

```
1  from my_module1 import x as m1x
2  print (m1x)
3  m1x = 11
4  print (m1x)
5
6
7
```

```
10
11
```

Where python will search for module? Firstly - program main dir, second %PATH%, third standard libraries, or 3rd party extensions

Modules & packages

Most interesting and useful python modules - index:

https://docs.python.org/3/library/index.html

Dunder methods & operator overloading

There are a number of methods with double underscore prefix and suffix.
They are called either **D**ouble **under**(score), or just magic methods. They're useful mainly for operators overloading - although can be used in that way only together with classes.

```
1  print ( 'a' + 'b' )
2  print ( __add__('a','b') )
3
4
5
6  ab
7  ab
```


```
1  def __add__(a,b):
2      print ("No way!")
3  print (1 + 2)
4
5
6  3
7
```

Dunder methods & operator overloading

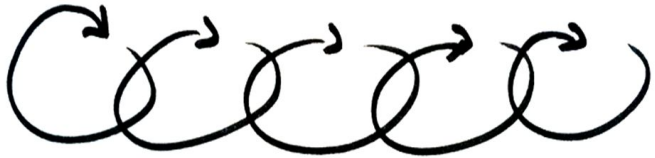
more in subject:
https://goo.gl/AUPq9Z
https://goo.gl/7wqJrt

```
1  class myclass():
2      def __add__(self,b):
3          return ”No way!”
4
5  Obj1 = myclass()
6  print (Obj1 + 2)
7
8
9  No way!
```

Iterators & generators

Iterator is an object created for traversing through entire collection. Consists of two methods: __iter__() and __next__(). To create Iterator use function **iter (collection_name)**.

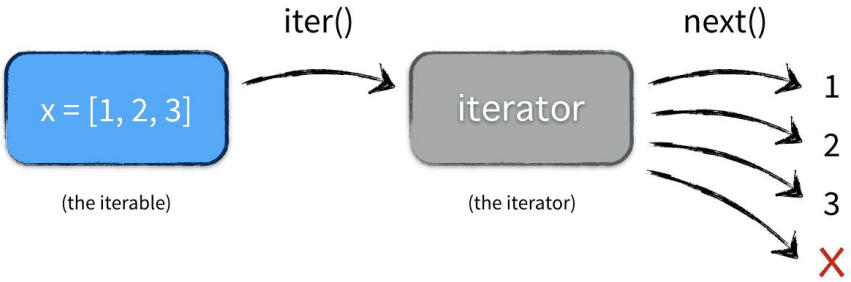
```
>>> mlist = ['a','b','c','d']
>>> it1 = iter(mlist)
>>> it1.__iter__()
<list_iterator object at 0x0000023C299E4128>
>>> it1.__next__()
'a'
>>> it1.__next__()
'b'
```



Iterators & generators

Generators is an easier way to create iterators using a keyword yield and defined with def keyword they can use several “yield” and return an iterator.

```
>>> def my_gen():
...     yield 1
...     yield 2
...     yield 3
...
>>> mlist = my_gen()
>>> mlist
<generator object my_gen at 0x0000023C299CACA8>
```



(cd on next page ->)

Iterators & generators

```
>>> mlist.__iter__()
<generator object my_gen at 0x0000023C299CACA8>
>>> mlist.__next__()
1
>>> mlist.__next__()
2
>>> mlist.__next__()
3
>>> mlist.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Decorators

Firstly, let's recap nested functions.

1. At line 20, function f1() is executed.
2. f1() returns f2(), which needs to be executed in the meantime.
3. f2() prints string, returns value (1)
4. In case we comment out line 17 - it will not return value (None).

```
1   def f1():
2       def f2():
3           print("this is f2")
4           return 2
5       return f2()
6
7
8   myvar = f1()
9   print (myvar) # NONE - 1
```

```
10  this is f2
    2
```

Decorators

We changed our plans a little:

- f2() doesn't return any value
- parenthesis from line 17 was deleted

Now we have access to f2() by using name myvar(). Let's add another function, this time outside f1().

```
1  def f1():
2      def f2():
3          print("this is f2")
4      return f2
5
6
7  myvar = f1()
8  print (myvar )
9
10
```

```
<function f1.<locals>.f2 at
0x0000022A66219620
```

Decorators

So we added f3(), outside other functions as promised.

What else did we change?
#1 - now f1() takes one argument, which is name of a function. It will be executed later.
#2 - executing function which was passed to f1()
#3 - New outside function
#4 - Now we want to pass f3() to f1 and finally:
#5 - execute f2(), by its new name - myvar.

Output is as clear as a day - f2() got executed, and from inside it executed f3() too. Unfortunately, f2 does not return anything

- that's why "None" came by.

```
1   def f1(f_inside): #1
2       def f2():
3           print ("this is f2")
4           f_inside() #2
5       return f2
6   def f3(): #3
7       print ("f3 here - outside")
8
9   myvar = f1(f3) #4
10  print ( myvar() ) #5
```

```
11  this is f2
12  f3 here - outside
    None
```

Decorators

Let's say we want to decorate our functions
in a more simple way.
Instead of executing f1 with another function
passed by argument (previous line 23),
we want to say that f3() is now part of f1()
and will be executed whenever we call it from f1()
(like in line 17)
That is one of decorator's jobs.

Definition of decorators:
"Function, which takes a function as a parameter,
and return function itself."

Read more:
https://goo.gl/CXvQBn


```
1   def f1(f_inside):
2       def f2():
3           print ("this is f2")
4           f_inside()
5       return f2
6   @f1
7   def f3():
8       print ("f3 here - outside")
9   print ( f3() )
```

```
10  this is f2
11  f3 here - outside
    None
```

File handling & context manager

```
1
2
3
4
5
6
7
8
9
```

```
fhandle = open('filename.ext','w')
str_ex = 'abc'
fhandle.write('typed:%s\n'%str_ex)

# (...)

fhandle.close()
```

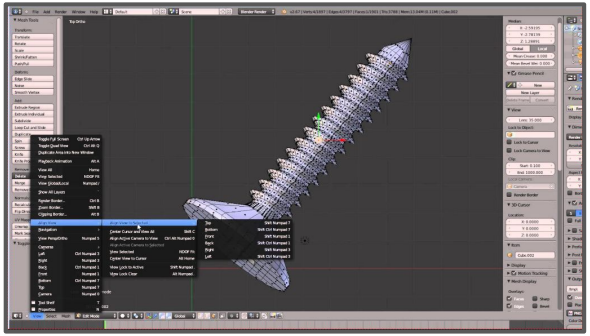
```
1
2
```

```
typed: abc
```

```
1
2
```

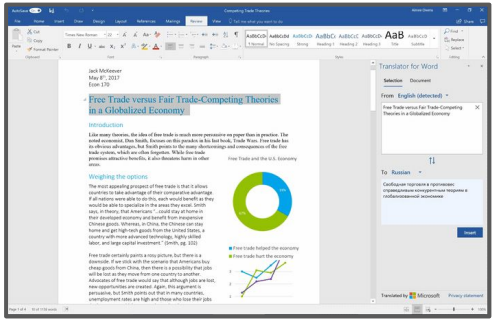
```
fhandle = open('fname.ext',
                'r / w / a ? ')
```

File handling & context manager



blender objects

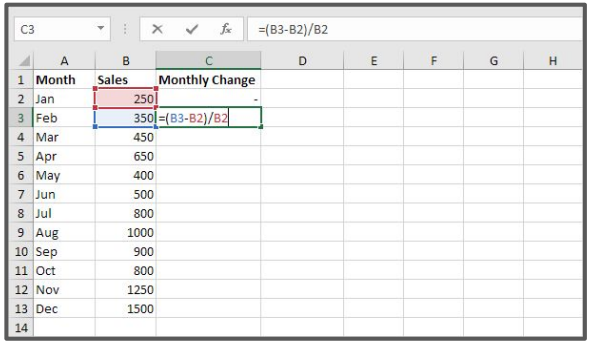
different kinds of files can be handled with python



word texts

audio streams

excel calcs



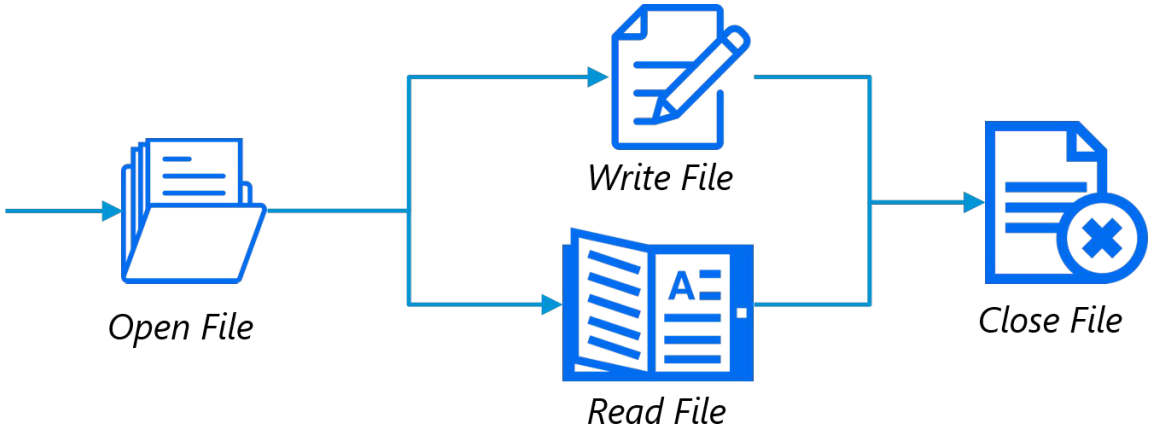
	A	B	C	D	E	F	G	H
1	Month	Sales	Monthly Change					
2	Jan	250	-					
3	Feb	350	=(B3-B2)/B2					
4	Mar	450						
5	Apr	650						
6	May	400						
7	Jun	500						
8	Jul	800						
9	Aug	1000						
10	Sep	900						
11	Oct	800						
12	Nov	1250						
13	Dec	1500						
14								

File handling & context manager

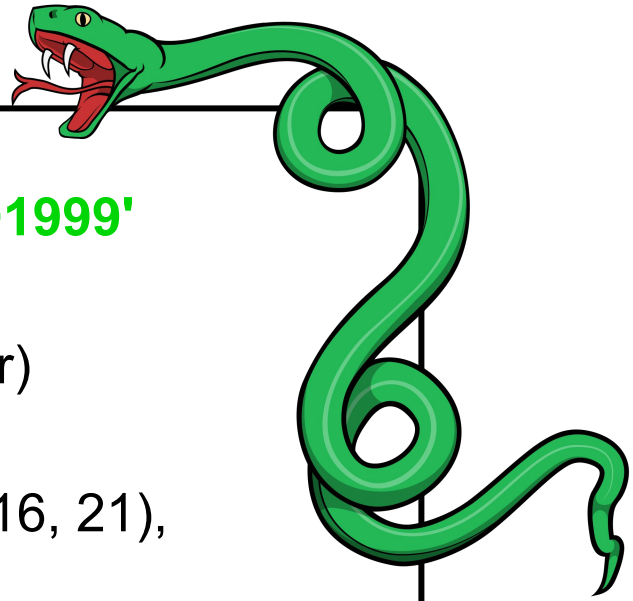
Always remember about a proper file handling. Each file handle processed in programs should be treated with all respect, so that not leaving any mess after visit.

For this purpose inventions like **context manager** are very useful. **Context manager** is a special class, which has defined several dunder methods for a proper file handling. That way, each time you need to work with files, it gives you security assurance and brings some comfort of file handling too.

Here's more:
https://goo.gl/5w6MJ5



Regular expressions



```
>>> import re
>>> mystr = 'abc aaa eefghAaaD1999'
>>>
>>> a = re.search(r'D.9{3}$',mystr)
>>> a
<_sre.SRE_Match object; span=(16, 21),
match='D1999'>
>>> a.group(0)
'D1999'
```

more: https://goo.gl/6GXEyv cheatsheet: https://goo.gl/Yp5HKq

Useful functions

about zip, map,filter, reduce and more:
https://goo.gl/eYyMXM

```
>>> l1 = [1,2,3,4]
>>> l2 = ['a','b','c','d']
>>> z1 = zip(l1,l2)
>>> z1
<zip object at 0x0000023C299E8888>
>>> dict(z1)
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>>
>>> m1 = map(lambda x: x**2, l1)
>>> list(m1)
[1, 4, 9, 16]
```

Quizizz

C:\Windows\system32\cmd.exe

```
Microsoft Windows [Version 10.0.17134.345]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\przem>
 >>>
 >>>
 >>> "Thanks for the attention"
```