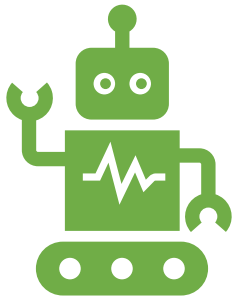
Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Вовед во Python 2



Аудиториски вежби по курсот
Вештачка интелигенција

2023/2024

Контрола на тек на извршување

if

```
x = 100
if x:  # equal to if(x != 0):
    print("1 - Got a true expression value.")
    print(x)
y = 0
if y:  # equal to if(y != 0):
    print("2 - Got a true expression value.")
    print(y)
    print("Good bye")
```

Внимавајте! Клучниот збор *if* се користи и во синтаксата за филтрирани *list comprehensions*.

Забелешка:

- Користете поравнување за блокови од наредби
- Две точки (:) после логички (boolean) изрази

if-else

```
x = 100
if x:  # equal to if(x != 0):
    print("Got a true expression value.")
    print(x)
else:
    print("Got a false expression value.")
    print(x)
    print("Good bye")
```

Забелешка:

- По клучниот збор *else* задолжително се поставуваат две точки (:)
- Може да има најмногу еден *else* блок придружен на дадена *if* наредба (*else* блокот е опционален)

if-elif-else

```
x = 2
if x == 3:
    print("X equals 3.")
elif x == 2:
    print("X equals 2.")
else:
    print("X equals something else.")
    print("This is outside the 'if'.")
```

- Во Python не постои наредба за избор од повеќе можности (како што е *switch* во C/C++)
- За ова се користи конструкцијата *if-elif-else*

Тернарен оператор

```
fruit = 'apple'
is_apple = True if fruit == 'apple' else False
```

- Тернарните оператори во Python се познати како условни изрази (conditional expressions).
- Овие оператори евалуираат нешто во зависност дали условот е вистинит или не.
- Дозволува брза проверка на условот, наместо повеќелиниски if израз.
- Најчесто може да биде значајно важен и може да го направи кодот компактен, а сепак одржлив.

while

```
>>> x = 3
>>> while x < 5:
        print(x, "still in the loop")
        x = x + 1
3 still in the loop
4 still in the loop
>>> x = 6
>>> while x < 5:
        print(x, "still in the loop")
>>>
```

while (2)

Може да дефинирате и else дел за while наредбата доколку завршувањето на циклусот треба да резултира во специфична операција

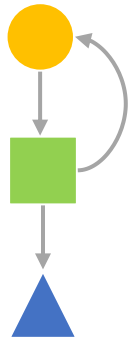
```
>>> x = 3
>>> while x < 5:
        print(x, "still in the loop")
        x = x + 1
    else: print(x, "out of the loop")
3 still in the loop
4 still in the loop
5 out of the loop
```

break и *continue*

- Можете да го искористите клучниот збор *break* во рамки на циклусот за целосно да излезете од *while*.
- Можете да го искористите клучниот збор *continue* во рамки на циклусот за да престанете со тековната итерација од циклусот и веднаш да преминете на следната.

```
>>> x = 0
>>> while x < 10:
        x += 1
        if x % 2 != 0:
            continue
        print(x, "even number")
        if x == 6:
            break
2 even number
4 even number
6 even number
```

For циклуси



For циклуси

- For циклусот ги изминува сите елементи од некоја колекција, или било каков друг податочен тип низ кој може да се итерира

```
for <item> in <collection>:
    <statements>
```

- Ако *<collection>* е листа или торка, тогаш *<statements>* се извршуваат за секој елемент на секвенцата.
- Ако *<collection>* е стринг, тогаш *<statements>* се извршуваат за секој знак во стрингот.

```
for char in "Hello world":
    print(char)
```

For циклуси (2)

```
for <item> in <collection>:
    <statements>
```

- <item> може да има и посложена структура од име на една променлива.
  - Кога и самите елементи на *<collection>* се некакви секвенци, тогаш *<item>* може да има иста структура како тие елементи.
  - Ваквото „повеќекратно“ доделување може да го олесни пристапот до поединечните делови на сложената структура на елементите.

```
>>> for (x, y) in [('a', 1), ('b', 2), ('c', 3), ('d', 4)]:
        print(x)
a
b
c
d
```

For циклуси и функцијата *range()*

- Во одредени случаи кога имаме потреба од променлива која ќе опфаќа одреден опсег (секвенца) од броеви, можеме да ја искористиме функцијата *range(lower,upper)* која враќа листа од броевите од *lower* до *upper*, но без *upper*.
  - *range(2,6)==[2,3,4,5]*
  - Функцијата враќа т.н. lazy секвенца, т.е. листата не се генерира сè додека не се наведе пристап до елемент на листата

```
>>> l = range(2, 6)
>>> l
range(2,6)
>>> for x in l:
        print(x)
2
3
4
5
```

  - ако *lower=0*, тогаш *range(0,upper)* може да се повика со само еден аргумент, *range(upper)*

- Во *range()* може да се дефинира и чекор на промена, т.е. која е разликата помеѓу два последователни елементи во генерираната лист
  - *range(lower,upper,step)*
  - *range(4,10,2)==[4,6,8]*

Алтернативен начин на изминување на секвенциски податочен тип

```
>>> fruits = ["banana", "apple", "mango"]
>>> fruits
["banana", "apple", "mango"]
>>> for index in range(len(fruits)):
        print(fruits[index])
banana
apple
mango
>>>
```

For циклуси и речници

```
>>> ages = {'Sam': 4, 'Mary': 3, 'Bill': 2}
>>> ages
{'Sam': 4, 'Mary': 3, 'Bill': 2}
>>> for name in ages.keys():
        print(name, ages[name])
Sam 4
Mary 3
Bill 2
>>>
```

pass…

- *pass*
  – Не прави апсолутно ништо

- Едноставно само „чува“ места за некој дел од код.
- Програмерите во Python обично го користат ако сакаат да „потрошат“ време на извршување или да означат дека треба на некое место да се доразвие кодот.

```
for i in range(1000):
    pass
```

Слично на празните загради {} во C++ or Java.

assert

- Наредбата *assert* се користи во текот на извршувањето за да се провери дали нешто е точно.
  - Ако условот е неточен, програмата запира
  - Попрецизно: програмата фрла исклучок - exception

```
assert(number_of_players < 5)
```

Генерирање листи со „List Comprehensions“

List = […]

List Comprehensions

```
[expression for name in list]
```

- Многу значајна програмска можност во Python.
  - Генерирање на нова листа преку примена на некоја функција врз секој елемент од оригинална листа.
  - Python програмерите интензивно ја користат оваа можност.
- Синтаксата за list comprehension е малку „незгодна“.
  - Комбинира *for*-циклус, *in* метод, и опционално *if*, но во рамки на list comprehension истите не се користат со стандардната синтакса.

List Comprehensions (2)

```
[expression for name in list]
```

```
>>> li = [3, 6, 2, 7]
>>> [elem * 2 for elem in li]
[6, 12, 4, 14]
```

- Каде *expression* е некаква пресметка или операција која ссе извршува врз променлива дадена со *name*.

- За секој елемент во *list*, list comprehension го прави следното:
  1. го поставува *name* да биде еднакво на тој елемент,
  2. ја пресметува новата вредност користејќи го *expression*,
- Потоа овие нови вредности ги собира во единствена листа која е резултатот (return вредноста) за list comprehension.

List Comprehensions (3)

```
[expression for name in list]
```

- Ако *list* содржи елементи од различни типови, тогаш *expression* мора да работи точно (синтаксички исправно) за секој можен тип кој го имаат елементите на *list*.
- Ако елементите од *list* се контејнери (комплексни типови), тогаш *name* може да биде контејнер (комплексен тип) од имиња кои би одговарале на типот и “обликот” на составните делови на елементите на *list*.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [n * 3 for (x, n) in li]
[3, 6, 21]
```

List Comprehensions (4)

```
[expression for name in list]
```

- Во рамки на *expression* може да се употребуваат и кориснички дефинирани функции.

```
>>> def subtract(a, b):
        return a - b
>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

Филтриран List Comprehension

```
[expression for name in list if filter]
```

- *Filter* одредува дали *expression* ќе се изврши врз секој елемент на *list*.
- За секој елемент на *list*, се проверува дали го задоволува условот *filter*.
- Ако условот врати *False*, тогаш тој елемент се изоставува од *list* пред да се евалуира изразот дефиниран за list comprehension.

Филтриран List Comprehension (2)

```
[expression for name in list if filter]
```

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Само 6, 7 и 9 го задоволуваат условот filter.
- Следствено, во резултатот се добива само 12, 14 и 18.

Вгнезден List Comprehension

- Благодарение на тоа што list comprehension изразите примаат листа на влез и продуцираат листа на излез, истите можат многу лесно да се вгнездуваат

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li]]
[8, 6, 10, 4]
```

- Вгнезденият list comprehension израз како резултат дава: [4, 3, 5, 2].
- Следствено, надворешниот дава: [8, 6, 10, 4].

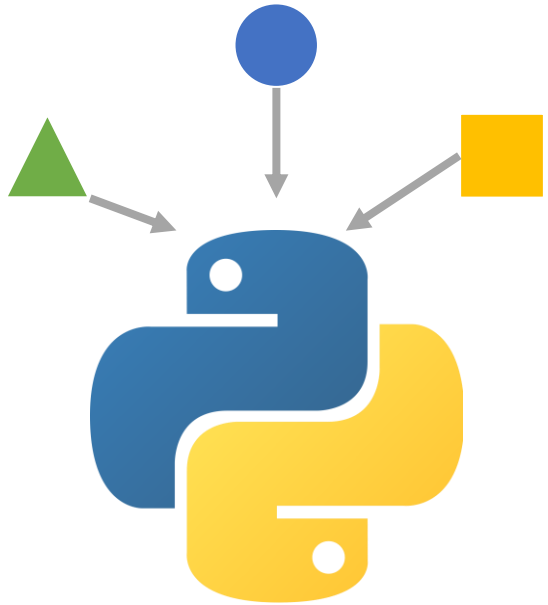
Вгнезден List Comprehension (2)

```
>>> nested = [[3, 'a', 9], [2, 7], [4, 1, 8, 'd']]
>>> flatten = [val for elem in nested for val in elem]
[3, 'a', 9, 2, 7, 4, 1, 8, 'd']
```

може да го замени

```
flatten = []
for elem in nested:
    for val in elem:
        flatten.append(val)
```

Импортирање и модули



Импортирање и модули

- Користење на класи и функции дефинирани во друга датотека.
- Во Python, модулот е датотека со соодветен Python код (.py екстензија)
- Слично на Java *import*, C++ *include*.
- Три облици на команда:

```
import somefile
from somefile import *
from somefile import className
```

- Што е разликата?
  - Што ќе се импортира од датотеката и како се пристапува до составните делови на модулот откако ќе се импортира.

import …

```
import somefile
```

- Целокупноста на somefile.py датотеката се импортира.
- За да се пристапи до нешто од датотеката, мора пред името на „нештото“ што сакаме да го искористиме да ставиме префикс за модулот т.е. „somefile“:

```
somefile.className.method("abc")
somefile.myFunction(34)
```

*from … import **

```
from somefile import *
```

- *Целокупноста* на somefile.py датотеката се импортира.
- За да се пристапи до нешто од датотеката, едноставно треба само да се наведе неговото име. Сè што се наоѓало во модулот сега се наоѓа во тековниот namespace.
- *Внимавајте!* Користењето на ваквиот облик на импортирање може лесно да ги пребрише дефинициите на локалните (постоечки) функции или променливи!

```
className.method("abc")
myFunction(34)
```

from … import ...

```
from somefile import className
```

- Само делот *className* од somefile.py датотеката се импортира.
- После импортирањето на *className* можете да го користите без префикс за модулот. Се наоѓа во тековниот namespace.
- *Внимавајте*! Ова може да ја пребрише дефиницијата на *className* доколку истата веќе постои во тековниот namespace!

```
className.method("abc")
myFunction(34)
```

← This is imported

← This is not imported

Вообичаено користени модули

Некои од често користените модули кои се импортираат, и се вклучени во стандардната инсталација:

- Модул: **sys** – многу корисни работи
- Модул: **os** – OS специфични кодови
- Модул: **os.path** – манипулација со патеки
- Модул: **math** – математички функции
- Модул: **random** – модул за случајни броеви
- Стандардна библиотека со модули за Python:
  - https://docs.python.org/3/library/index.html

Директориуми за модули

Каде ги бара Python модул датотеките?

- Листата на директориуми во кои Python ќе се обиде да ги најде модулите кои треба да се импортираат се наоѓа во: `sys.path`

(Променлива со име „path“ во модулот „sys“.)

- За да додадете директориум каде се наоѓаат ваши модули треба таа патека да ја „прилепите“ на постоечката листа.

```
sys.path.append('/my/new/path')
```

Простори на имиња (namespaces)

Простор на имиња (namespace)

- Во Python, променливата е име (идентификатор) кое што референцира некој објект
- **Простор на имиња** (анг. namespace) претставува речник што се состои од имиња на променливи (клучеви) и објекти (вредности) кои што им се придружени на нив
- Секоја функција и секоја класа си има свој локален простор на имиња
- Дадена наредба во Python може да пристапува имиња од *локален простор на имиња* или од *глобалниот простор на имиња*

Локална и глобална променлива

Ако локална и глобална променлива имаат исто име, локалната променлива „фрла сенка на“ (ја препокрива) глобалната променлива.

```
>>> vkupno = 0  # Ova e globalna promenliva
>>> def zbir(x, y):  # Definicija na funkcija
        vkupno = x + y  # Ova e lokalna promenliva
        print("Vo funkcijata, vkupno = ", vkupno)
        return vkupno
>>> zbir(10, 20)
>>> print("Nadvor od funkcijata, vkupno = ", vkupno)

Vo funkcijata, vkupno = 30
Nadvor od funkcijata, vkupno = 0
```

Пристап до глобални променливи во тело на функција

- Python прави едуцирани претпоставки за тоа дали променливите се локални или глобални
- Тој претпоставува дека секоја променлива на која што и се доделува вредност во рамки на телото на некоја функција е локална променлива
- Затоа, за да се додели вредност на глобална променлива во телото на некоја функција неопходно е да се употреби наредбата *global*
- Наредбата *global x* му укажува на Python дека x е глобална променлива
  - Веќе не се пребарува локалниот простор на имиња за оваа променлива!

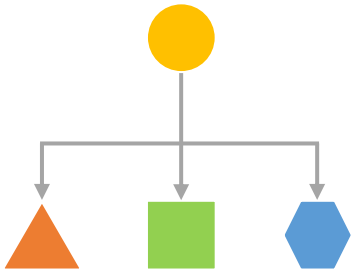
Пристап до глобални променливи во тело на функција (2)

```
>>> Money = 2000
>>> def add_money():
        global Money
        Money = Money + 1

>>> print(Money)
>>> add_money()
>>> print(Money)

2000
2001
```

Пристап до глобални променливи во тело на функција (3)

- Променливата ***Money*** е дефинирана во глобалниот простор на имиња
- Во функцијата ***add_money()***, на променливата ***Money*** и доделуваме вредност, па Python претпоставува дека ***Money*** е локална променлива
  – Затоа ние му најавуваме дека *Money* е глобалната променлива дефинирана претходно – користејќи ја наредбата *global Money*
  – Без наредбата *global Money*, кодот ќе резултираше со *UnboundLocalError* (грешка заради пристап до непостоечка локална променлива)

Објектно-ориентирано програмирање во Python



Сè е објект…

- Сè во Python претставува објект!
  - Од она што претходно го видовме…

```
"hello".upper()
list3.append('a')
dict2.keys()
```

  - Овие примери изгледаат како повици на методи во Java или C++.
  - Покрај ваквите вградени податочни типови, може да се дефинираат и нови податочни типови (класи).
- Програмирањето во Python вообичаено ја користи објектно-ориентираната парадигма.

Дефинирање на класа

- *Класа* е специјален податочен тип кој дефинира како да се изградат определен тип на објекти.
  - *Класата* дополнително чува и одредени податочни ставки кои ги споделуваат сите инстанци од дадената класа.
  - *Инстанци* се објекти кои се креирани врз основа на дефинициите дадени во рамки на класата.
- Python не користи посебен интерфејс за дефинирање на класи како во други јазици. Овде едноставно ја дефинирате класата и веднаш можете да ја користите.

Методи во класите

- *Методи* во рамки на *класа* се дефинираат преку вклучување на дефиниции на функции во рамки на блокот за дефинирање на класата.
  - Во рамки на секоја дефиниција на метода првиот аргумент мора да биде `self`, којшто претставува референца што се врзува со инстанцата која ја повикува методата
  - Во најголем број на класи вообичаено е да постои метода со име `__init__`

Пример на едноставна класа за студент

```
class Student:
    """Klasa za reprezentacija na student."""
    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
```

Инстанцирање на објекти

- Во Python не постои клучен збор како „new“ во Java!
- Едноставно го користите името на класата придружено со мали загради - (), во кои ги наведувате соодветните аргументи кои го градат објектот и тоа го доделувате на некоја променлива
- `__init__()` е конструкторот за класата – метода што се повикува секогаш кога се создава објект од таа класа. Вообичаено прави одредени иницијализации.
- Аргументите кои се наведуваат во заградите после името на класата се аргументите со кои се повикува `__init__()` методата.
  - За да креираме инстанца од класата Student со име Marko и возраст од 21 година пишуваме:

```
b = Student("Marko", 21)
```

Конструктор: __init__()

- `__init__` методата може да има произволен број на аргументи.
  - Како и за секоја друга функција, така и за конструкторот аргументите можат да се дефинираат со подразбирани вредности, со што стануваат опционални во повикот.

- Сепак, првиот аргумент мора да биде `self`

Self

- Првиот аргумент во секоја метода е референца кон тековната инстанца на класата.
  - По конвенција, овој аргумент се нарекува ***self***.
- Во `__init__`, ***self*** се однесува на објектот кој тековно се креира, додека кај другите методи на инстанцата која ја повикала методата.
  - Слично на *this* во Java или C++.
  - Но, во Python употребата на *self* е почеста отколку употребата на *this* во Java.
- При повик на која било метода не треба да се наведува ***self*** како прв аргумент – овој аргумент е подразбирлив
  - Истото важи и при инстанцирање на објект (т.е. при повик до конструкторот)

Бришење на инстанци

- Кога ќе завршите со работа со одреден објект, не мора експлицитно да го бришете или ослободувате.
  - Python има автоматско „**собирање на ѓубре**“ (анг. *garbage collection*).
  - Python автоматски детектира кога повеќе нема референци (не може да се пристапи) кон одреден дел од меморијата и автоматски го ослободува овој дел.
  - Генерално работи многу добро и поретко се случуваат проблеми со меморијата.
  - Класата може да имплементира посебен метод *__del__()*, наречен деструктор, кој се повикува кога инстанцата треба да се уништи. Овој метод може да се користи за да се исчистат било кои немемориски ресурси употребени од инстанцата.

Да се потсетиме: дефиницијата на Студент

```
class Student:
    """Klasa za reprezentacija na student."""
    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
```

Традиционална синтакса за пристап

```
>>> f = Student("Bob Smith", 23)

>>> f.full_name      # Access an attribute.
"Bob Smith"

>>> f.get_age()      # Access a method.
23
```

Пристап до непознати делови од класата

- Проблем: Понекогаш името на некој атрибут или метода во рамки на класа се добива во време на извршување (run time)…

- Решение: `getattr(object_instance, string)`
  - **`string`** е стринг кој содржи име на атрибут или метода од класа
  - **`getattr(object_instance, string)`** враќа референца кон тој атрибут или метода

getattr(object_instance, string)

```
>>> f = Student("Bob Smith", 23)

>>> getattr(f, "full_name")
"Bob Smith"

>>> getattr(f, "get_age")
<bound method Student.get_age of <__main__.Student object
at 0x0000013D8436B5C0>>

>>> getattr(f, "get_age")()     # We call this.
23

>>> getattr(f, "get_birthday")
AttributeError: 'Student' object has no attribute 'get_birthday'
```

hasattr(object_instance, string)

```
>>> f = Student("Bob Smith", 23)

>>> hasattr(f, "full_name")
True

>>> hasattr(f, "get_age")
True

>>> hasattr(f, "get_birthday")
False
```

Два типа на атрибути

- Во дадена класа, сè што не е метода е **атрибут**.

- *Податочни атрибути*
  - Променлива која припаѓа на *конкретна инстанца* од класата.
  - Секоја инстанца има посебна вредност за овој тип на атрибут.
  - Вообичаен тип на атрибути.

- *Класни атрибути*
  - Припаѓаат на *класата како целина*.
  - *Сите инстанци на класата ја споделуваат истата вредност*.
  - Во некои јазици се нарекува ***static*** променлива.
  - Добри за:
    - константи на ниво на класа
    - бројач на инстанци креирани од класата

Податочни атрибути

- Податочните атрибути се креираат и иницијализираат со конструкторот `__init__()`.
  - Со доделување вредност на некое име на атрибут истиот се креира.
  - Во рамки на класата, податочните атрибути се референцираат (пристапуваат) со наведување на префиксот **`self`**.
    - на пример, **`self.full_name`**

```
class Teacher:
    """Klasa za reprezentacija na nastavnik."""
    def __init__(self,n):
        self.full_name = n

    def print_name(self):
        print(self.full_name)
```

Класни атрибути

- Поради фактот дека сите инстанци на една класа делат единствена копија на класниот атрибут важи:
  - ако *било која* инстанца ја промени вредноста на класниот атрибут, таа се менува за *сите* инстанци.
- Класните атрибути се дефинираат
  - *во рамки* на блокот за дефиниција на класата
  - *надвор* од дефинициите на методите
- Поради тоа што даден класен атрибут е единствен за цела класа, а не единствен за секоја инстанца, пристапот до овие атрибути е поинаков:
  - Нотацијата за пристап до класен атрибут е `self.__class__.name`
    - (Ова е еден од можните начини, и најбезбеден во општ случај)

```
class Sample:
    x = 23

    def increment(self):
        self.__class__.x += 1
```

```
>>> a = Sample()
>>> a.increment()
>>> a.__class__.x
24
```

Податочни vs. Класни атрибути

```
class Counter:
    overall_total = 0  # class attribute

    def __init__(self):
        self.my_total = 0  # data attribute

    def increment(self):
        Counter.overall_total = \
            Counter.overall_total + 1
        self.my_total = \
            self.my_total + 1
```

```
>>> a = Counter()
>>> b = Counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Наследување: надкласа и подкласа

- Една класа може да ја *прошири* дефиницијата на некоја друга класа
  - Дозволува користење (или проширување) на методите и атрибутите во класата која се проширува
  - По конвенција, новата класа се нарекува *подкласа*, а постоечката - *родител*, *предок* или *надкласа*
- За да дефинирате подкласа, потребно е во првата линија на дефиницијата после името на подкласата во загради да го наведете името на надкласата.

```
class AIStudent(Student):
```

  - Python не користи некој посебен клучен збор за да се означи наследство.
  - Нуди поддршка за повеќекратно наследство.

```
class AIStudent(Student, Person):
```

Пример за наследување

```
class Parent:  # Definicija na klasata - roditel
    parent_attr = 100  # Klasen atribut

    def __init__(self):
        print("Calling parent constructor")

    def parent_method(self):
        print("Calling parent method")

    def set_attr(self, attr):
        Parent.parent_attr = attr

    def get_attr(self):
        print("Parent attribute:", Parent.parent_attr)
```

```
class Child(Parent):  # Definicija na klasata - dete
    def __init__(self):
        print("Calling child constructor")

    def child_method(self):
        print("Calling child method")
```

```
>>> c = Child()
Calling child constructor
>>> c.child_method()
Calling child method
>>> c.parent_method()
Calling parent method
>>> c.set_attr(200)
>>> c.get_attr()
Parent attribute: 200
```

Редефинирање на методи

- За да се *редефинира метода* од надкласата, во рамки на подкласата треба да се наведе дефиниција на метода со истото име.
  - При повик се извршува новата дефиниција.

- За да се изврши метода од надкласата во ситуација кога имате нова дефиниција за таа метода во подкласата, потребно е да направите експлицитен повик на методата од надкласата (со префикс за надкласата).

```
ParentClass.method_name(self, a, b, c)
```

  - Ова е единствениот случај кога во повик на метода експлицитно го проследувате `self` како аргумент (повик на истоименувана метода од надкласа).

Дефиниција на проширување на класата за студент

```
class Student:
    """Klasa za reprezentacija na student."""

    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
```

```
class AIStudent(Student):
    """Klasa koja proshiruva student."""

    def __init__(self, n, a, s):
        super(AIStudent, self).__init__(n, a)  # Call __init__ for Student
        self.section_num = s

    def get_age(self):  # Redefines get_age method entirely
        print("Age: " + str(self.age))
```

Проширување на конструкторот __init__

- Исто како за било која друга метода…
  - Вообичаено во рамки на подкласата се извршува `__init__` методата на надкласата, како и останати потребни команди.
  - Во рамки на дефиницијата на конструкторот на подкласа често ќе го сретнете следното:

```
ParentClass.__init__(self, x, y)
```

каде `ParentClass` се однесува на името на надкласата.

super() функција

- Како кај другите објектно-ориентирани јазици, се дозволува повик на методи на надкласата од подкласата.
- Два типични употреби на super:
  - Во класната хиерархија со еднократно наследство, super може да се користи за да се посочи на родителските класи без да се именуваат експлицитно, со што кодот станува поодржлив.

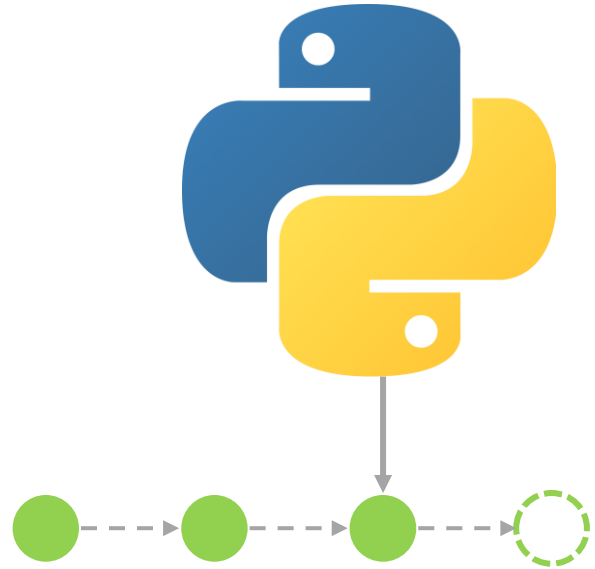
```
super().__init__()  # Calling parent constructor

super(ChildClass, self).__init__()  # Calling parent constructor
```

  - Подршка за кооперативно повеќекратно наследство во околина на динамичко извршување.
    - Овој случај е уникатен за Python и не се наоѓа кај статички компајлираните јазици или јазици кои не подржуваат повеќекратно наследство.
    - Ова овозможува да се имплементираат „дијамант дијаграми“ кадешто повеќе основни класи ја имплементираат истата метода.
    - Добар дизајн диктира дека ваквата метода има иста дефиниција за повик во секој случај.

Приватни податоци и методи

- Атрибутите или методите кои имаат две водечки долни црти __ во своето име, но истите ги немаат на крајот од името, се приватни. До истите не може директно да се пристапи надвор од класата.
  - *Забелешка 1*:
    Атрибутите и методите со две долни црти и на почетокот и на крајот од името се однесуваат на вградени методи и атрибути за класата.
  - *Забелешка 2*:
    Не постои „protected“ тип во рамки на Python; следствено, до приватните податоци немаат пристап ниту подкласите.

Python итератори и генератори



Слични на секвенци, но…

- Итераторите се слични на секвенците (листи, торки), но…
- Не ја добивате наеднаш целата секвенца
- Елементите се добиваат еден по еден, во моментот кога ќе бидат потребни и според тоа како се дефинирани
- Можат да се дефинираат бесконечни секвенци (пример, сите позитивни броеви)
- Можете да креирате ваши итератори, ако дефинирате функција која го генерира следниот елемент

Датотеките се итератори

```
>>> f = open("myfile.txt")
>>> for line in f.readlines():
      print(len(line))
9
21
35
43
>>> f = open("myfile.txt")
>>> for line in f:
      print(len(line))
9
21
35
43
```

readlines() враќа листа од линиите во датотеката

Датотеката е итератор, враќа вредности според спецификацијата

Датотеките се итератори

- Итераторите можат да се користат во било кое сценарио каде можете да итерирате низ колекции (пример: листи, торки, речници, ...)

```
>>> f = open("myfile.txt")
>>> map(len, f.readlines())
[9, 21, 35, 43]
>>> f = open("myfile.txt")
>>> map(len, f)
[9, 21, 35, 43]
```

Пример: fib.py

```
class FibNum:
    def __init__(self):
        self.fn2 = 1
        self.fn1 = 1

    def __next__(self):  # next() е срцето на секој итератор
        # следното доделување е ефикасно не само од аспект
        # на кратење на линии од код, но и од аспект да се осигураме
        # дека ќе ги искористиме само старите вредности
        # на self.fn1 и self.fn2 во доделувањето
        (self.fn1, self.fn2, old_fn2) = (self.fn1 + self.fn2,
                                         self.fn1, self.fn2)
        return old_fn2

    def __iter__(self):
        return self
```

next() се користи за добивање на следни вредности

Класи со __iter__() метода се итератори

Пример: fib.py (2)

```
>>> from fib import *

>>> f = FibNum()
>>> for i in f:
      print(i)
      if i > 100:
            break
1
1
2
3
...
144
```

Автоматско запирање на итератор

```
class FibNum20:
    def __init__(self):
        self.fn2 = 1
        self.fn1 = 1

    def __next__(self):
        (self.fn1, self.fn2, old_fn2) = (self.fn1 + self.fn2,
                                         self.fn1, self.fn2)
        if old_fn2 > 20:
            raise StopIteration
        return old_fn2

    def __iter__(self):
        return self
```

Покренете ја оваа грешка за да го запрете итераторот

Автоматско запирање на итератор (2)

```
>>> from fib import *

>>> for i in FibNum20():
      print(i)
1
1
2
3
5
8
13
```

Нешто дополнително

- Функцијата **`list`** ги материјализира вредностите на итераторот како листа

```
>>> list(FibNum20())
[1, 1, 2, 3, 5, 8, 13]
```

- **`sum()`**, **`max()`**, **`min()`** можат да работат со итератори

```
>>> sum(FibNum20())
33
>>> max(FibNum20())
13
>>> min(FibNum20())
1
```

itertools

- Модулот **`itertools`** содржи корисни алатки за работа со итератори
- **`islice()`** е функција со која можете да земате делови од итераторот

```
>>> from itertools import *

>>> list(islice(FibNum(), 6))
[1, 1, 2, 3, 5, 8]
>>> list(islice(FibNum(), 6, 10))
[13, 21, 34, 55]
```

- Уште многу корисни функции …

Python генератори

- Python генераторите продуцираат итератори
- Тие се воопштување и се помоќни од итераторите
- Дефинирате функција и наместо да користите return за да генерирате вредност, ја користите yield наредбата
- Во моментот кога е потребна следна вредност, генератор функцијата продолжува од каде што застанала
- За крај покренете StopIteration или повикајте **return**

Пример за генератор

```
def gy():
    x = 2
    y = 3
    yield x, y, x + y
    z = 12
    yield z / x
    yield z / y
    return
```

```
>>> g = gy()
>>> print(next(g))
(2, 3, 5)
>>> print(next(g))
6.0
>>> print(next(g))
4.0
>>> print(next(g))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

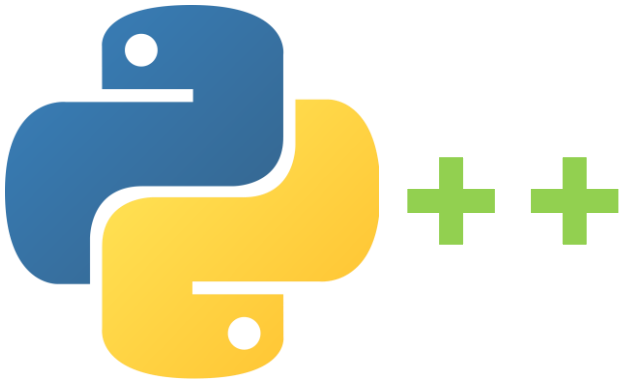
Сите броеви на Фибоначи: fib()

```
def fib():
    fn2 = 1
    fn1 = 1
    while True:
        (fn1, fn2, old_fn2) = (fn1 + fn2, fn1, fn2)
        yield old_fn2
```

Сите зборови во датотека: get_word()

```
def get_word(fl):
    for line in fl:
        for word in line.split():
            yield word
    return
```

конечно крај ... или не?



Процесирање на датотеки со Python

Едноставен пример:

```
file_ptr = open('filename')
some_string = file_ptr.read()
for line in file_ptr:
    print(line)
file_ptr.close()
```

```
with open('filename') as file_ptr:
    some_string = file_ptr.read()
    for line in file_ptr:
        print(line)
```

Со клучниот збор **`with`** се дефинира блок на код во кој дадена датотека е отворена, а потоа надвор од блокот датотеката е затворена.

За детали можете да ја погледнете документацијата на Python
https://docs.python.org/3/library/filesys.html

Исклучоци во Python

```
try:
    1 / 0
except:
    print('That was silly!')
finally:
    print('This gets executed no matter what')
```

```
'That was silly!'
'This gets executed no matter what'
```