Whats and whys for

beginners

by Justas Trimailovas @ VilniusPy

github.com/trimailov
j.trimailovas@gmail.com

Myself and Python

learnt basics of C from university,
programming seemed arduous

Myself and Python

2 years ago
discovered Python through Udacity's MOOC

and I loved it!

Myself and Python

little by little advancing through Python, Linux and growing interest in various technologies

About this talk

"5 WTFs in Python"

About this talk

answer questions raised by novices in Python
(or in programming in general)

About this talk

novice me

answer questions raised by ~~novices~~ in Python
(or in programming in general)

About this talk

novice me

answer questions raised by ~~novices~~ in Python
(or in programming in general)

*questions are not in any particular order

About this talk

things you can learn from a lazy guy,
so you can improve yourself little faster

Not in this talk

Python 101

How to setup your machine for Python development

`vim`, `emacs`, Sublime Text, PyCharm is best Python IDE

Not in this talk

~~Python 101~~

How to setup your machine for Python development

`vim`, `emacs`, Sublime Text, PyCharm is best Python IDE

Not in this talk

~~Python 101~~

~~How to setup your machine for Python development~~

`vim`, `emacs`, Sublime Text, PyCharm is best Python IDE

Not in this talk

~~Python 101~~

~~How to setup your machine for Python development~~

~~`vim, emacs,` Sublime Text, PyCharm is best Python IDE~~

Not in this talk

~~Python 101~~

~~How to setup your machine for Python development~~

~~vim, emacs, Sublime Text, PyCharm is best Python IDE~~

Though you can freely ask about all that later

How can software have so many files?

```
> tree .
.
└── first
    ├── baz.py
    ├── functions
    │   ├── __init__.py
    │   └── bar.py
    └── other_functions
        ├── __init__.py
        └── biz.py
```

WHAT?! empty?!

```
> cat first/functions/bar.py

def drink_beer(where):
    print(“fun @ %s” % where)
```

```
> cat first/baz.py

from functions import bar

bar.drink_beer(where=“snekutis”)
```

What is `self`?

```python
class Character:
    def __init__(self, int=1, chr=1, str=1):
        self.int = int  # intelligence
        self.chr = chr  # charisma
        self.str = str  # strength

    def learn(self):
        self.int += 1

    def socialize(self):
        self.chr += 1

    def train(self):
        self.str += 1
```

```python
class Character:
    def __init__(self, int=1, chr=1, str=1):
        self.int = int  # intelligence
        self.chr = chr  # charisma
        self.str = str  # strength

    def learn(self):
        self.int += 1

    def socialize(self):
        self.chr += 1

    def train(self):
        self.str += 1

    def get_self(self):
        print(self)
```

```
>>> knight = Character()
>>> knight.get_self()
>>> <Character object at 0x106160a20>
```

`self` is an instance of a class

Method

```
class Character:
    ..
    def learn(self):
        self.int += 1
    ..
```

Function

```
..
def learn(character):
    character.int += 1
..
```

Instance of a method of is passed automatically, but not received

Instance of a method of is passed automatically, but not received

Received	Passed

`class Character:`
`  ..`
`  def learn(self):`
`  self.int += 1`
`  ..`
	`knight.learn()`

Instance of a method of is passed automatically, but not received

Received

```
class Character:
    ..
    def learn(self):
        self.int += 1
    ..
```

Passed

```
knight.learn()
```

Why so many

`__underscores__`?

It’s a Python convention used for builtin attributes and to avoid namespace conflicts

More conventions exist

```
_internal_use
```

```
__very_private -> _classname__very_private
```

```
>>> dir(knight)
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'chr',
 'int',
 'learn',
 'socialize',
 'str',
 'train']
```

__call__ - method called on instance call
__new__ - method called on instance creation (before __init__)
__init__ - method for object initialization
__dict__ - get attribute:value dictionary
__doc__ - return docstring
__eq__ - describes '==' operator
__lt__ - describes '<' operator, etc.

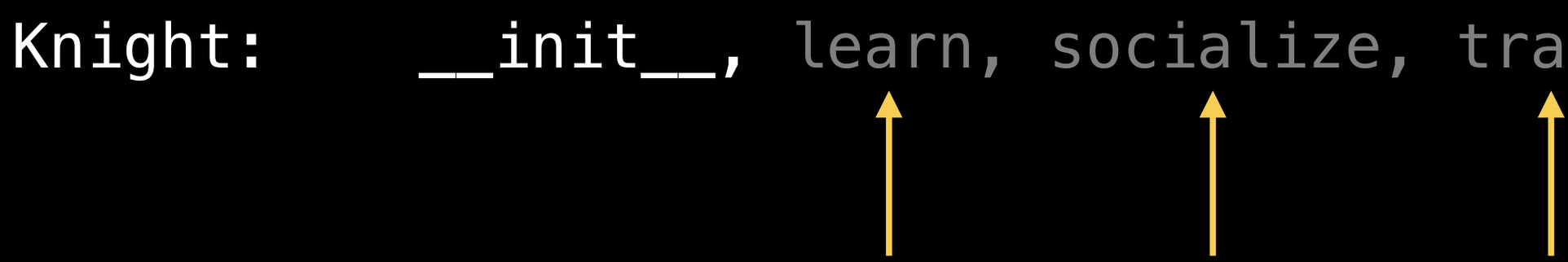
https://docs.python.org/3.4/reference/datamodel.html

What is `super`?

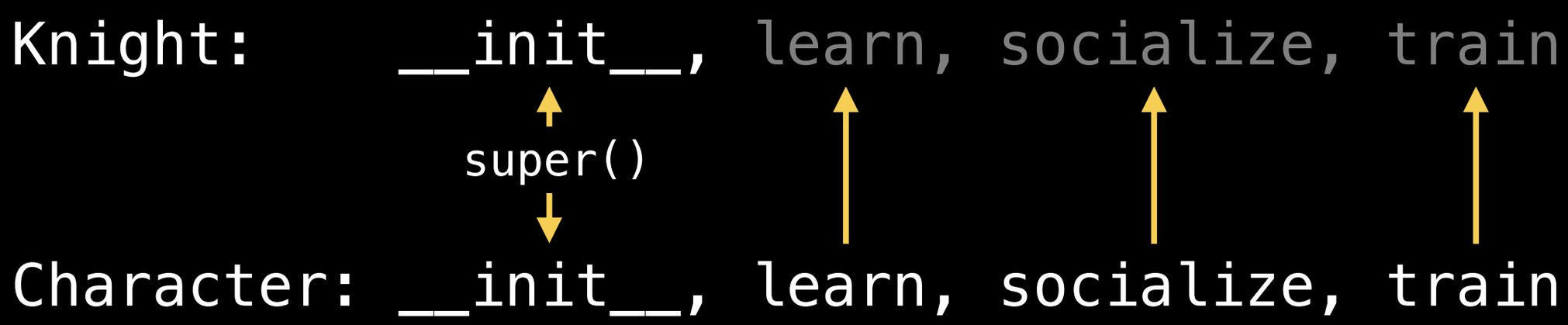
```
class Knight(Character):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.race = “Human”
```

If `super()` would not have been used, child `class` would have overriden it's parent's `__init__` method.

Then initialized Knight would have `race`, but no `int`, `str` or `chr`.



`super()` let's us call same method from parent `class`.
This way we can very flexibly extend classes.



*Python allows multiple inheritance, same principles apply, though some cases can be trickier.
Reading about Python's Method Resolution Order (MRO) is a must.

But what is this magic with *args and **kwargs?

But what is this magic with `*args` and `**kwargs`?

*in my head it sounds like *arcs and quarks*

* - unwraps positional arguments

** - unwraps keyword arguments

* - unwraps positional arguments

** - unwraps keyword arguments

You can name them what ever you like, e.g. `*arcs`, `**quarks`.
Though `*a`, `**kw` and similar are usual.

Basically what it means, you can let function to accept arbitrary arguments

```
class Knight(Character):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.race = “Human”

baby_knight_params = {‘int’: 0,
                      ‘chr’: 5,
                      ‘str’: 15}

knight = Knight(**baby_knight_params)
```

```
class Character:
    def __init__(self, *args, **kwargs):
        ..
        self.hp = kwargs.get(“hp”, 50)
```

When you're lazy,
as *args and **kwargs are just shorter

```
def ugly_function(*args, **kwargs):
    do_ugly_stuff_with(args, kwargs)

def get_ugly_params():
    random_args = call_api()
    sorted_args = sort_args(random_args)

    params_as_dict = call_dict_api()

    ugly_function(*sorted_args, **params_as_dict)
```

What is `yield`?

`yield` returns a generator object

Function

```
def simple():
    l = []
    for i in range(5):
        l.append(i)
    return i
```

Generator

```
def simple():
    for i in range(5):
        yield i
```

Saves memory
Lazy evaluation
Could be endless (RNG)
generator.__next__()

Function

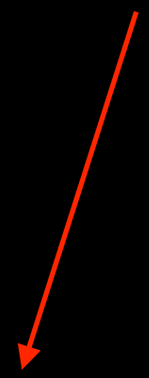
```
def simple():
    l = []
    for i in range(5):
        l.append(i)
    return i
```

Generator

```
def simple():
    for i in range(5):
        yield i
```

Tips and tricks

What this code does?



```
class Knight(Character):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.race = “Human”
```

Answer: good editor and ctags

good editor - can read tags files or generate them. By using shortcut, we can jump to class/method definition

ctags - software which indexes your code

Bonus answer: `ag`, `ack`, `grep`

```
>>> ag ‘class Character’
```

What this code does at runtime?

Answer: `pdb` and `ipdb`

```
import ipdb; ipdb.set_trace()
             or
 import pdb; pdb.set_trace()
```

Conclusions and recomendations

Hold your horses: read manuals, documentations, sources

Python and brief C knowledge let's me appreciate and enjoy programming

Try to understand how things work in general way, no need to guess

Low level knowledge helps