Python Batteries - oTree Concepts - Tutorial #0

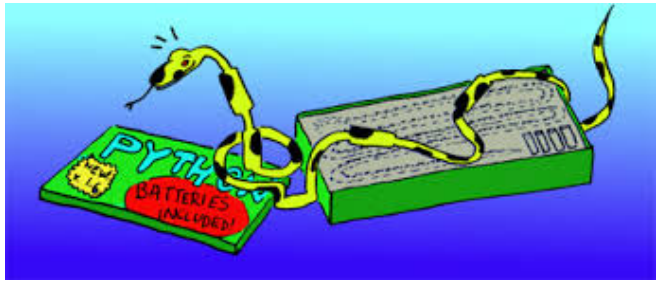
Juan Cabral - jbc.develop@gmail.com

Jan, 2018

Python Batteries

- The Python source distribution has long maintained the philosophy of **"batteries included"** – having a rich and versatile standard library which is immediately available, without making the user download separate packages. This gives the Python language a head start in many projects.
- **However, the standard library modules aren't always the best choices for a job.**
- How many functionalities have the python batteries? Try:

```
import antigravity
```



Python Batteries - os module

This module provides a portable way of using operating system dependent functionality.

- The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface.
- Extensions peculiar to a particular operating system are also available through the os module, but using them is of course a threat to portability.
- If not separately noted, all functions that claim "Availability: Unix" are supported on Mac OS X, which builds on a Unix core.
- Docs: https://docs.python.org/3/library/os.html

os module - Basic operations

- operative system family

```
# check platform module for more details
>>> os.name
'posix'
```

- The current directory

```
>>> os.getcwd()
'/home/jbcabral/projects/otree_bogota2018/src'
```

os module - Basic operations

- create a new directory with the name “name”

```
>>> os.mkdir("name")
```

- create directory `name` and their parent `parent`

```
>>> os.makedirs("parent/name")
```

- remove file (also check the module `shutil`)

```
>>> os.remove("filename")
```

os module - Waliking though files

- List a directory

```
>>> os.path.listdir("path")
```

- Recursive listing

```
>>> for root, dnames, fnames in os.walk(path):
        for fname in fnames:
            print(os.path.join(root, fname))
```

os module - Paths

- Check if a path is a file

```
>>> os.path.isfile("/home/juan/.bashrc")
True
```

- Check if a path is a directory

```
>>> os.path.isdir("/home/juan/.bashrc")
False
```

- Check if a path exists

```
>>> os.path.exists("/home/juan/.bashrc")
True
```

os module - Path of the current module

```
PATH = os.path.abspath(os.path.dirname(__file__))
```

- Every module has a attribute called `__file__` with the relative path of the current file.
- `os.path.dirname` remove the "file" part of the `__file__` attribute.
- `os.path.abspath` convert the path to an absolute path.

Python Batteries - `datetime` module

The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.

- Full Documentation: https://docs.python.org/3/library/datetime.html

datetime module: Basic

```
import datetime
now = datetime.datetime(2003, 8, 4, 12, 30, 45)

print(now)
# => 2003-08-04 12:30:45

print repr(now)
# => datetime.datetime(2003, 8, 4, 12, 30, 45)

print type(now)
# => <type 'datetime.datetime'>

print(now.year, now.month, now.day)
# => 2003 8 4

print(now.hour, now.minute, now.second)
# => 12 30 45
```

datetime module: Convert from another type

```
import datetime
import time

print(datetime.datetime(2003, 8, 4, 21, 41, 43))
# => 2003-08-04 21:41:43

datetime.datetime.today()

datetime.datetime.now()

datetime.datetime.fromtimestamp(time.time())

datetime.datetime.utcnow()

datetime.datetime.utcfromtimestamp(time.time())
```

datetime module: to String

```
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2017, 12, 5, 23, 37, 53, 112972)

>>> now.ctime()
'Tue Dec  5 23:37:53 2017'

>>> now.isoformat()
'2017-12-05T23:37:53.112972'

>>> now.strftime("%Y%m%dT%H%M%S")
'20171205T233753'
```

datetime module: Algebra

```
>>> past = datetime.datetime.now()
>>> past
datetime.datetime(2017, 12, 5, 23, 37, 53, 112972)


>>> datetime.datetime.now() - past
datetime.timedelta(0, 26, 329369)

>>> datetime.datetime.now() - past
datetime.timedelta(0, 27, 864966)

>>> datetime.datetime.now() - past
datetime.timedelta(0, 29, 289356)
```

datetime module: Algebra 2

```
>>> past = datetime.datetime.now()
>>> past
datetime.datetime(2017, 12, 5, 23, 42, 54, 209964)


>>> delta = datetime.datetime.now() - past
datetime.timedelta(0, 24, 168670)

>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2017, 12, 5, 23, 43, 55, 510720)

>>> now + delta
datetime.datetime(2017, 12, 5, 23, 44, 19, 679390)
```

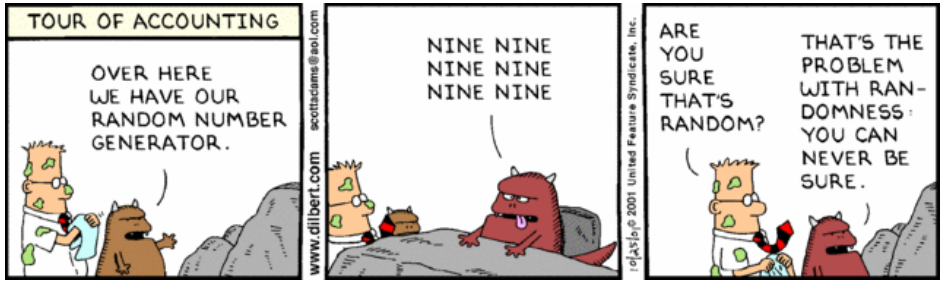
`datetime` module

Resume:

- The `datetime.datetime` type represents a date and a time during that day.
- The `datetime.date` type represents just a date, between year 1 and 9999
- The `datetime.time` type represents a time, independent of the date.
- The `datetime.timedelta` type represents the difference between two time or date objects.
- The `datetime.tzinfo` type is used to implement timezone support for time and datetime objects (this will not be cover in this tutorial).
- In servers always use **utcnow()**

Python Batteries - `random` module

- This module implements pseudo-random number generators for various distributions.
  For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.
  There are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions.
- Full Documentation:
  https://docs.python.org/3/library/random.html



random module: basic

```
>>> import random

# Random float:  0.0 <= x < 1.0
>>> random.random()
0.37444887175646646

# Random float:  2.5 <= x < 10.0
>>> random.uniform(2.5, 10.0)
3.1800146073117523

# Interval between arrivals averaging 5 seconds
>>> random.expovariate(1 / 5)
5.148957571865031

# Integer from 0 to 9 inclusive
>>> random.randrange(10)
7
```

random module: basic 2

```
# Even integer from 0 to 100 inclusive
>>> random.randrange(0, 101, 2)
26

# Single random element
>>> random.choice(['win', 'lose', 'draw'])
'draw'

# Shuffle a list
>>> deck = 'ace two three four'.split()
>>> random.shuffle(deck)
>>> deck
['four', 'two', 'ace', 'three']

# Four samples without replacement
>>> random.sample([10, 20, 30, 40, 50], k=4)
[40, 10, 50, 30]
```

random module: Simulations

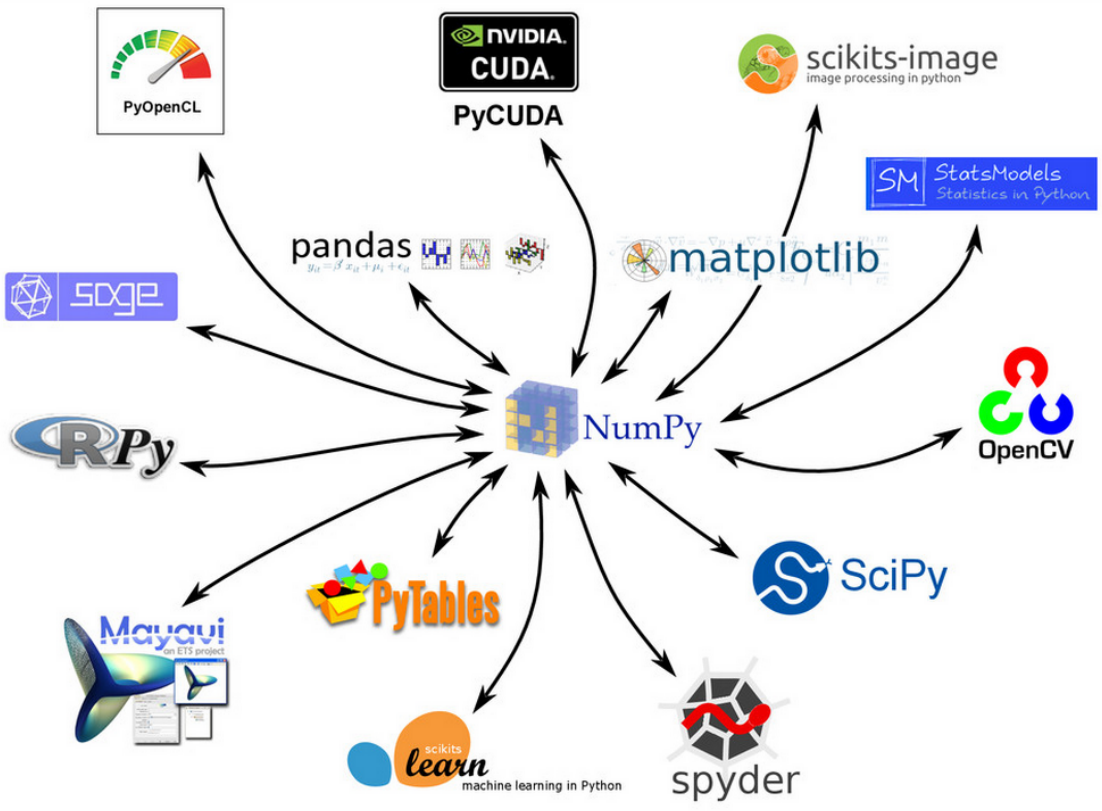
```
>>> import collections
>>> import random

# Six roulette wheel spins
# (weighted sampling with replacement)
>>> random.choices(['red', 'black', 'green'],
                   [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

# Deal 20 cards without replacement from a deck of
# 52 playing cards and determine the proportion of
# cards with a ten-value (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = random.sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15
```

random module: Simulations 2

```
>>> import random
>>> import statistics

# Estimate the probability of getting 5 or more heads
# from 7 spins of a biased coin that settles on
# heads 60% of the time.
>>> def trial():
        return random.choices(
                'HT', cum_weights=(0.60, 1.00),
         k=7).count('H') >= 5
>>> statistics.mean(trial() for _ in range(10000))
0.4169
```

`random` module: resume

- `random` and `statistics` are useful in many cases but is not feature complete.
- In complex cases you must install numpy and scipy



Python Batteries - `itertools` module

- This module implements a number of iterator building blocks
- Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.
- Full Documentation: https://docs.python.org/3/library/itertools.html

Python Batteries - `itertools` module

- Repeat something n times

```
>>> for e in itertools.repeat(10, 3):
...     print(e)
10
10
10
```

- Repeat any iterable forever

```
>>> list_cycle = itertools.cycle([1,2,3])
>>> next(list_cycle)
1
>>> next(list_cycle)
2
>>> next(list_cycle)
3
>>> next(list_cycle)
1
```

Python Batteries - `itertools` module

- Simple filter

```
>>> for e in itertools.compress('ABCDE', [1,0,1,0,1]):
...     prin(e)
A
C
E
```

- Iterable concatenation

```
>>> for e in itertools.chain('A', 'DEF'):
...     print(e)
A
D
E
F
```

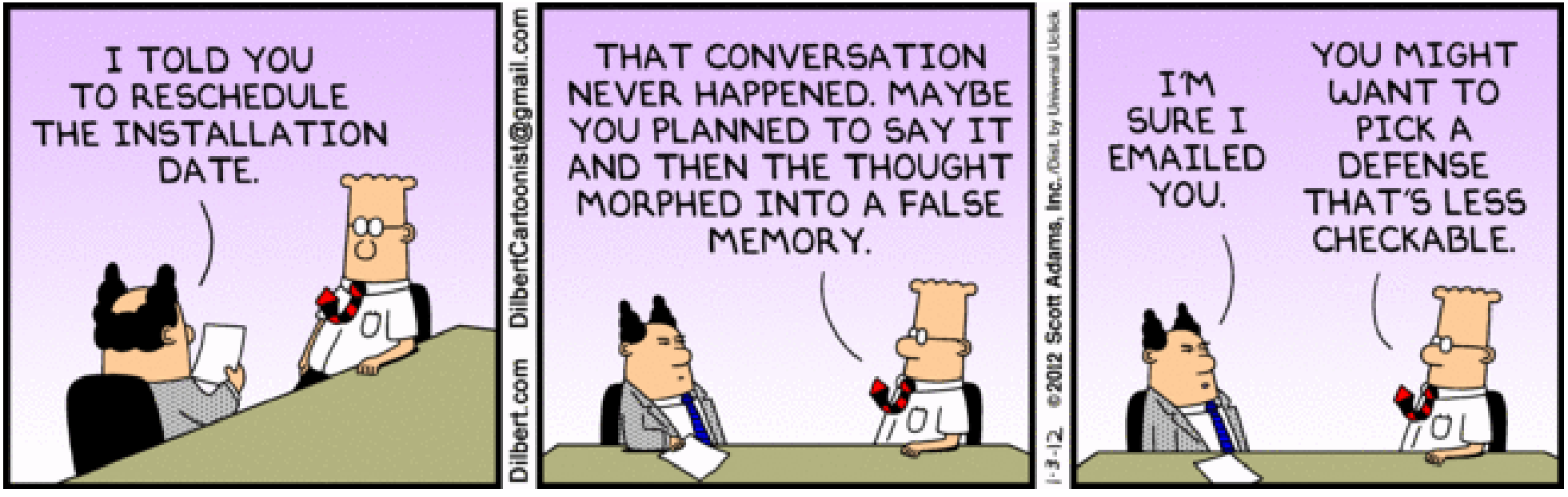
Python Batteries - `itertools` module

Combinatoric generators:

Iterator	Arguments	Results
`product()`	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements
`product('ABCD', repeat=2)`		`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`		`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`		`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`		`AA AB AC AD BB BC BD CC CD DD`

Python Batteries - `pip` module and tool

pip is a package management system used to install and manage software packages written in Python. Many packages can be found in the Python Package Index (PyPI)



IT'S A COMMAND LINE TOOL!!!

Python Batteries - pip search

```
$ pip search otree
otree-boto2-shim (0.3.2)          - ...
otree-save-the-change (2.0.0)     - Automatically ...
otree-core (1.4.29)               - oTree is a ...
otree-custom-export (0.0.4)       - customizing...
otree-dulwich-windows (1.0)       -
hiwi (0.1)                        - Integrate ...
mongotree (0.1.3)                 - Python ...
mturkotreeutils (0.0.3)           - set ...
otree (0.1)                       -
scikit-otree (0.5)                - oTree ...
otree-redwood (0.6.6)             - oTree...
otreechat (0.2.1)                 - oTree chat.
otreeutils (0.2.3)                - A package...
ovmm (0.2.2)                      - ovmm manages...
slider-task (0.1.1)               - oTree Slider Task.
```

Python Batteries - `pip` install

- Install

```
$ pip install otree-core
```

- Install Legacy version

```
$ pip install otree-core==1.4
```

- Upgrade

```
$ pip install -U otree-core
```

oTree

A modern open platform
for social science experiments

oTree

oTree is a **framework** based on Python and Django that lets you build:

- **Homepage:** http://www.otree.org/
- Multiplayer strategy games, like the prisoner's dilemma, public goods game, and auctions
- Controlled behavioral experiments in economics, psychology, and related fields
- Licensed under the **MIT open source license** with the added requirement of a citation of the paper.

quotation{ > Chen, D. L., Schonger, M., & Wickens, C. (2016). oTree-An open-source platform for laboratory, online, and field experiments. Journal of Behavioral and Experimental Finance, 9, 88-97.}

oTree Concepts

Sessions

In oTree, a session is an event during which multiple participants take part in a series of tasks or games. An example of a session would be:

> *"A number of participants will come to the lab and play a public goods game, followed by a questionnaire. Participants get paid EUR 10.00 for showing up, plus their earnings from the games."*

oTree Concepts

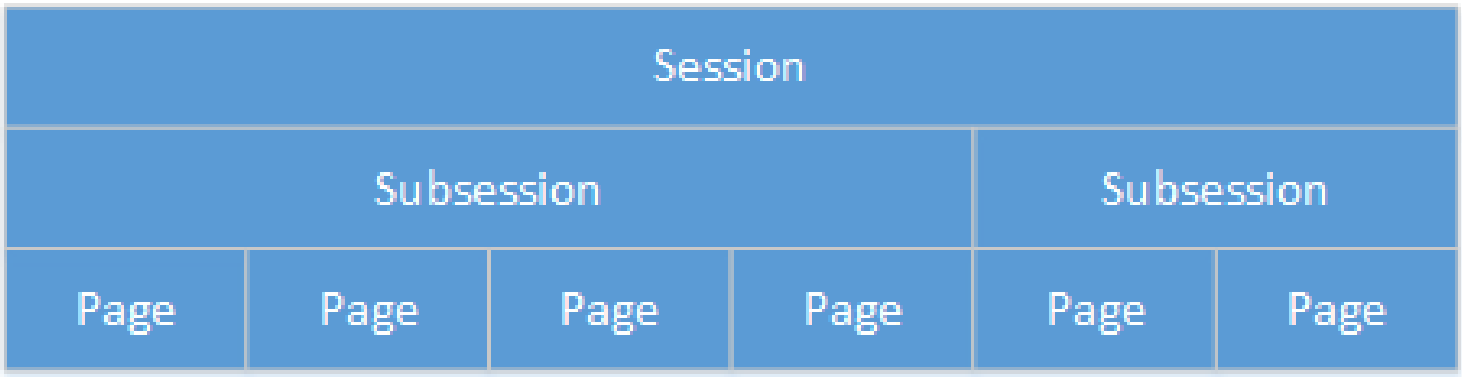
Subsessions

A session is a series of subsessions; subsessions are the "sections" or "modules" that constitute a session. For example:

> *if a session consists of a public goods game followed by a questionnaire: - the public goods game would be subsession 1 - and the questionnaire would be subsession 2.*

In turn, each subsession is a sequence of pages the user must navigate through. For example:

> *if you had a 4-page public goods game followed by a 2-page questionnaire:*



oTree Concepts

Groups

Each subsession can be further divided into groups of players; for example:

> *you could have a subsession with 30 players, divided into 15 groups of 2 players each. (Note: groups can be shuffled between subsessions.)*

oTree Concepts

Object hierarchy

oTree's entities can be arranged into the following hierarchy:

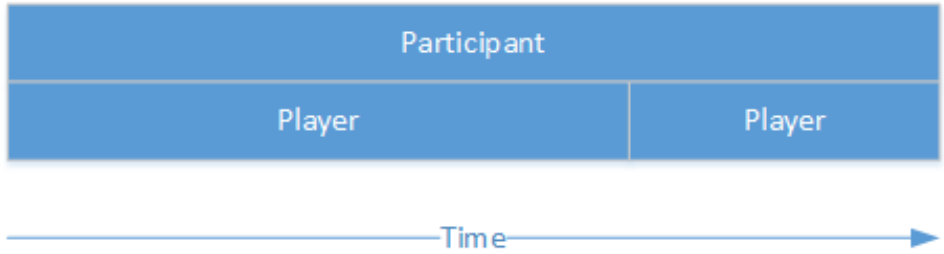
```
Session
  |
  +-Subsession
      |
      +-Group
          |
          +-Player
              |
              +-Page
```

- A session is a series of subsessions
- A subsession contains multiple groups
- A group contains multiple players
- Each player proceeds through multiple pages

oTree Concepts

Participant

In oTree, the terms "player" and "participant" have distinct meanings. The relationship between participant and player is the same as the relationship between session and subsession:



A player is one participant in one particular subsession. A player is like a temporary "role" played by a participant. A participant can be player 2 in the first subsession, player 1 in the next subsession, and so on.

Enough Talk!

- Open your console (Powershell, terminal, or any flaored pyton console)
- Open an editor (PyCharm, SublimeText, Kate, Atom. . .)
- Follow Me!

What is “self”?

In Python, self is an instance of the class you’re currently under. If you are ever wondering what self means in a particular context, scroll up until you see the name of the class.
In the below example, self refers to a Player object:

```
class Player(object):

    def my_method(self):
        return self.my_field
```

In the next example, however, self refers to a Group object:

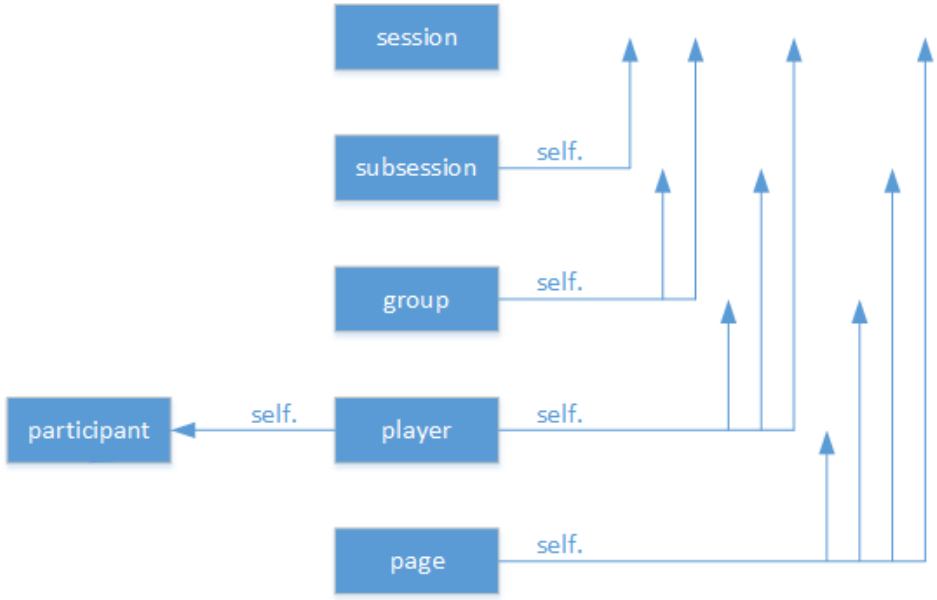
```
class Group(object):

    def my_method(self):
        return self.my_field
```

What is “self”?

- self is conceptually similar to the word **“me”**.
  - You refer to yourself as “me”,
  - but others refer to you by your name.
  - And when your friend says the word “me”, it has a different meaning from when you say the word “me”.

What is “self”?

Here is a diagram of how you can refer to objects in the hierarchy within your code:



For example, if you are in a method on the `Player` class, you can access the player's payoff with `self.payoff` (because `self` is the player). But if you are inside a `Page` class in `views.py`, the equivalent expression is `self.player.payoff`, which traverses the pointer from ‘page’ to ‘player’.

`self` Examples - oTree-core

```
class Session(...) # defined in oTree-core
    def example(self):

        # current session object
        self

        self.config

        # child objects
        self.get_subsessions()
        self.get_participants()
```

self Examples - oTree-core

```
class Participant(...) # defined in oTree-core
    def example(self):

        # current participant object
        self

        # parent objects
        self.session

        # child objects
        self.get_players()
```

self examples - in your models.py

```
class Subsession(BaseSubsession):
    def example(self):

        # current subsession object
        self

        # parent objects
        self.session

        # child objects
        self.get_groups()
        self.get_players()

        # accessing previous Subsession objects
        self.in_previous_rounds()
        self.in_all_rounds()
```

self examples - in your models.py

```
class Group(BaseGroup):
    def example(self):

        # current group object
        self

        # parent objects
        self.session
        self.subsession

        # child objects
        self.get_players()
```

self examples - in your models.py

```
class Player(BasePlayer):
    def my_custom_method(self): ...

    def example(self):
        # current player object
        self

        # method you defined on the current object
        self.my_custom_method()
        # parent objects
        self.session
        self.subsession
        self.group
        self.participant
        self.session.config

        # NEXT SLIDE
```

self examples - in your models.py

```
class Player(BasePlayer):

    def my_custom_method(self):
        ...

    def example(self):

        # PREVIOUS SLIDE

        # accessing previous player objects
        self.in_previous_rounds()

        # self.in_previous_rounds() + [self]
        self.in_all_rounds()
```

self examples - in your views.py

```
class MyPage(Page):
    def example(self):

        # current page object
        self

        # parent objects
        self.session
        self.subsession
        self.group
        self.player
        self.participant
        self.session.config
```

Finally How to Code

Most programming languages follow a basic style or formatting standard to make it easy for others to read your code. In Python, we have the PEP 8 and the PEP 20 conventions. PEP stands for "Python Enhancement Proposal".

- **PEP 8 – Style Guide for Python Code**
  https://www.python.org/dev/peps/pep-0008/
- **PEP 20 – The Zen of Python**
  https://www.python.org/dev/peps/pep-0020/

PEP 20

Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one –obvious way to do it.

PEP 20

- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

PEP 8 - Code lay-out

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python.

- Always use four spaces to indent code. Do not use tabs, tabs introduce confusion and are best left out.
- Wrap your code so that lines don't exceed 79 characters. This helps users with small displays and makes it possible to have several code files open side by side on larger displays.
- When vertical aligning text, there should be no arguments on the first line

PEP 8 - Code lay-out

```
# Good:
my_list = [
    1, 2, 3
    4, 5, 6,
]

foo = long_function_name(
    var_one, var_two,
    var_three, var_four)

# Bad:
foo = long_function_name(var_one, var_two,
    var_three, var_four)

result = some_function_that_takes_arguments(
'argument one',
'argument two', 'argument three')
```

PEP 8 - Whitespace.

- Use 2 blank lines around top level functions and classes.
- Use 1 blank line to separate large blocks of code inside functions.
- 1 blank line before class method definitions.
- Avoid extraneous whitespace.
- Use blank lines sparingly.
- Always surround binary operators with a space on either side but group them sensibly.
- Don't use spaces in keyword arguments or default parameter values.
- Don't use whitespace to line up operators.
- Multiple statements on the same line are discouraged.
- Avoid trailing whitespace anywhere

PEP 8 - Whitespace.

```
# Good:
x = 1
y = 2
long_variable = 3

# Bad:
x             = 1
y             = 2
long_variable = 3
```

PEP 8 - Whitespace.

```
# Good:
if x == 4: print x, y
x, y = y, x
spam(ham[1], {eggs: 2})

# Bad:
if x == 4 : print x , y
x , y = y , x
spam( ham[ 1 ], { eggs: 2 } )
```

PEP 8 - Whitespace.

```
# Good:
i = i + 1
c = (a+b) * (a-b)
hypot2 = x*x + y*y

# Bad:
i=i+1
c = (a + b) * (a - b)
hypot2 = x * x + y * y
```

PEP 8 - Whitespace.

```
# Good:
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()

# Bad:
if foo == 'blah': do_blah_thing()
do_one; do_two(); do_three()
```

PEP 8 - Comments

- Comments should be complete sentences in most cases.
- Keep comments up to date
- Write in “Strunk & White “English
- Inline comments should be separated by at least two spaces from the statement and must start with ‘#’ and a single space.
- Block comments should be indented to the same level as the code that follows them.
- Each line in block comments starts with ‘#’.
- Write docstrings for all public modules, functions, classes and methods.
- Docstrings start and end with “”" e.g “”" A Docstring. “”“.
- Single line docstrings can all be on the same line.
- Docstrings should describe the method or function’s effect as a command.
- Docstrings should end in a period.
- When documenting a class, insert a blank line after the docstring.
- The last “”" should be on a line by itself

PEP 8 - Comments

```
def kos_root():
    """Return the pathname of KOS root directory."""
    global _kos_root
    blah
    blah

def a_complex_function(parameter=False):
    """
    A multiline docstring.

    Keyword arguments:
    parameter -- an example parameter (default False)

    """
```

PEP 8 - Imports

- Don't use wildcards.
- Try to use absolute imports over relative ones
- Don't import multiple packages per line
- When using relative imports, be explicit with (.)

PEP 8 - Imports

```
# Good:
import os
import sys
from subprocess import Popen, PIPE

import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example

from . import sibling
from .sibling import example

# Bad:
import os, sys # multiple packages
import sibling # local module without "."
from mypkg import * # wildcards
```

PEP 8 - Naming Conventions

- Use CapWords/CamelCase convention for class names.
- Use short, lowercase identifiers separated by underscores for modules and functions.
- Always use “self” for the first argument to instance variables.
- Always use “cls” for the first argument to class methods.
- Constants should be in ALL_CAPS_WITH_UNDERSCORES.

Finally

Clean code means less problems (and money)

References

- https://goo.gl/Dnc7SF
- https://pip.pypa.io
- https://docs.python.org/3/library/shutil.html
- https://docs.python.org/3/library/statistics.html
- https://docs.python.org/3/library/collections.html
- https://docs.python.org/3/library/itertools.html
- https://docs.python.org/3/library/os.html
- https://docs.python.org/3/library/random.html
- https://docs.python.org/3/library/time.html
- https://docs.python.org/3/library/datetime.html
- https://www.python.org/dev/peps/pep-0206/
- http://otree.readthedocs.io/en/latest/