# Programming Paradigms in Python

Johan Falkenjack

February 5, 2018

# Programming Paradigms

- An overarching philosophy about programming
  - Execution model
  - Code organization
- Most famous paradigms:
  - Imperative (write commands to change *state*)
  - Procedural (group commands into *subroutines*)
  - Declarative (commands describe *what* to do, *not how*)
  - Functional (organize code as *stateless* functions)
  - Logic (describe computation as a set of *facts* and *rule-relations*)
  - Object-oriented (organize code as objects with both *local state and the procedures to manipulate that state*)
  - Symbolic (allow a language to *change itself*)

# Programming Paradigms

## Combinations

- Some paradigms can be adhered to at the same time
  - Imperative-Procedural (Ada, etc.)
  - Declarative-Functional (Lisp, etc.)
  - Declarative-Logical (Prolog, etc.)
- Most paradigms can be combined in a single program
- Few languages today adhere strictly to any one or a set of paradigms

# Programming Paradigms

## Paradigms in Python

- Python is a multi-paradigm language
- Mix of:
  - Imperative
  - Procedural
  - Functional
  - Object-oriented

# Functional Programming

- Features functional programming
- Examples from Kris Jenkins
- Higher-order functions
- How to handle state? Pass it around.

# Functional Programming

## General features

- Pure functions: Functions with no side-causes and no side-effects.
- First-class functions: Functions are not treated as different from other data, it is simply data which can be executed.
- Higher-order functions: Functions which act on other functions, taking them as input or returning them as output.
- Referential transparency: As a result of pure functions, calling function A with input x will always yield output y, no matter when A is called.
- Mathematical thinking: Functional programming is the paradigm most similar to Math, especially algebra.

# Pure functions

## Explicit input and output

- "I put it to you that every function you write has two sets of inputs and two sets of outputs." - Kris Jenkins
- Think of the state of a program before calling a function as a hidden set of inputs.
- Think of the state changes performed by the function as a hidden set of outputs.
- Hidden complexity.
- Functional programming is about not hiding such complexity.

# Pure functions

## What about encapsulation?

- Encapsulation is hiding complexity so that the programmer doesn't need to think about it.
- Doesn't functional programming throw out the baby with the bathwater here?
- Encapsulation is about hiding the implementation details of the code.
- Encapsulation is *not* about hiding how the code interacts with the outside world.

# Pure functions

## No side causes

```python
from datetime import datetime, timedelta


def fiveMinutesFromNow():
    return datetime.now() + timedelta(minutes=5)

def fiveMinutesFrom(t):
    return t + timedelta(minutes=5)
```

# Pure functions

## No side effects

```python
from datetime import datetime, timedelta
```

```python
def stamp():
    ts = datetime.now() - timedelta(seconds=1)
```

```python
def stamp(t):
    return t - timedelta(seconds=1)

ts = stamp(datetime.now())
```

# First-class functions

## Functions are just executable data

- In Python, a function is an (immutable) object like any other.
- Function objects have the `__call__` function (more on this in the OOP section)
- Functions can be put in containers such as lists, etc.
- Just like with other types, functions can be assigned to variables, but are not in themselves named.
- The `def` statement implicitly ties a function to a variable name.
- The `lambda` statement creates an anonymous function object.
  - Though limited to one-liners.

# First-class functions

## Examples

```
fns = [fiveMinutesFrom,
       fiveMinutesFromNow]

fns[1](datetime.now())

pure = {fiveMinutesFrom:"Pure",
        fiveMinutesFromNow:"Impure"}

hello = lambda name: "Hello,␣{}!".format(name)
```

# Higher-order functions

## Functions on functions

- "Regular" which don't operate on other functions are First-order functions.
- All other functions are Higher-order functions.
- Functions as input, output or both.
- Bread and butter of functional programming.
- Most famous: `map`, `filter`, `reduce`

# Higher-order functions

## Standard Higher-order functions

- `map`
  - Takes a 1-parameter function and a collection (e.g. list, tuple) and applies the function to each element in the collection.
  - Returns a map object with the results.
- `filter`
  - Takes a 1-parameter function and a collection (e.g. list, tuple) and applies the function to each element in the collection.
  - Returns the collection with all the elements for which the function returned True.
- `reduce`
  - Takes a 2-parameter function, a collection (e.g. list, tuple) and an optional initial value and recursively replaces the first element in the collection with the result of calling the function on the first two elements.
  - Returns the result of the innermost recursive call.

# Examples

## map

```
In [1]: map(lambda x: x**2, (1, 2, 3, 4))
Out[1]: <map at 0x7faaf8b5e9b0>

In [2]: tuple(map(lambda x: x**2, (1, 2, 3, 4)))
Out[2]: (1, 4, 9, 16)

In [3]: list(map(lambda x: x**2, (1, 2, 3, 4)))
Out[3]: [1, 4, 9, 16]
```

# Examples

## filter

```
In [1]: filter(lambda x: x % 2 == 0,
   ...:        (1, 2, 3, 4))
   ...:
Out[1]: <filter at 0x7faaf8b5ecf8>

In [2]: tuple(filter(lambda x: x % 2 == 0,
   ...:              (1, 2, 3, 4)))
   ...:
Out[2]: (2, 4)
```

# Examples

## reduce

```
In [1]: from functools import reduce

In [2]: reduce(lambda x, rest: x * rest,
   ...:            (1, 2, 3, 4))
   ...:
Out[2]: 24

In [3]: reduce(lambda x, rest: (x, rest),
   ...:            (1, 2, 3, 4))
   ...:
Out[3]: (((1, 2), 3), 4)

In [4]: reduce(lambda x, rest: (x, rest),
   ...:            (1, 2, 3, 4), 0)
   ...:
Out[4]: ((((0, 1), 2), 3), 4)
```

# Higher-order functions

## Functions returning functions

- As functions are like any other object in Python, they can be returned from other functions
- Best known built-in example is `partial`
  - Takes a function `fn` and set of named arguments to `fn`.
  - Returns a function which acts as `fn` but with the named parameters bound to the values of the named arguments to `partial`.
  - Creates a function of fewer parameters and avoids possibly expensive reevaluation of arguments which remain constant over many calls to `fn`.

# Closures

- When a function is returned it can access the environment in which it was created.
- This can be exploited to introduce a small measure of state into functional programming.
- Changing a variable in a closure requires declaring it `nonlocal`.

# Closures

## Example

```python
def make_greeter(n):
    def hello():
        return "Hello,␣my␣name␣is␣{}.".format(n)
    return hello

def make_switch(state):
    def switch():
        nonlocal state
        state = not state
        return state
    return switch
```

# Object-Oriented Programming

## General philosophy

- Mutable data has its uses.
- State is often necessary.
- But we still want to relatively strictly control side-causes and side-effects.
- Okay, we'll allow data to change, but data can only change itself.

# Object-Oriented Programming

## General features

- Classes: A datatype with data and the functions to access and manipulate that data combined.
- Inheritance: A class can *inherit* features from another class (sub-classing).
- Instantiation: To create an *instance*, an object, of a given class.
- Poly-morphism: Interaction works the same for different sub-classes of the same class even though data and behavior might differ.

# OLSReg

## An example class

- Imagine we have a class called OLSReg with the member functions `train`, `coefficients` and `predict`.
  - The `train` function takes a set of training examples and fits an Ordinary Least Squares Regression to the data.
  - The `coefficients` returns the fitted coefficients of the OLS model.
  - The `predict` function takes a new observation and predicts the outcome.

# OLSReg

## A usage example

```
model = OLSReg()

model.train(data)

model.coefficients()

model.predict(obs)
```

# The dot notation

- The *dot notation* (e.g. `object.function()`) indicates that in the namespace of an object, there exists a name (which might be tied to a function) which we want to access.
- You encountered this notation in the first lab, for instance when using the `format` function in the string class.

# OLSReg

## A class definition example

```
class OLSReg(object):

    def __init__(self):
        self.coeff = None

    def train(self, dataset):
        # Code to fit coefficients to
        # dataset using OLS
        ...

    def coefficients(self):
        return self.coeff

    def predict(self, newobs):
        return product(self.coeff, newobs)
```

# Instances and instance variables

- The `__init__` function is a special function called a constructor which creates a new instance of a class, it is called when we execute `OLSReg()`.
- The `self` parameter refers to the instance of the class on which the function is currently called (or which has just been created, in the case of `__init__`.
- As `model` is an instance of `OLSReg`, `model.coefficients()` can be thought of as `OLSReg.coefficients(model)` where `self` becomes the object `model` itself.
- The variable `self.coeff` is referred to as an instance variable as it is connected to a specific instance of the class and thus might differ between different instances.
- Any function taking `self` as first argument is considered an *instance function* or *instance method*.

# Class functions and class variables

- Some functions and variables might relate to the class itself rather than a specific instance.
- Class variables are defined on the global level of the class and are shared between all instances.
- Naturally, the functions are referred to as *class functions* or *class methods* and the variables as class variables.
- When defining a class function which needs access to class variables or other class functions, use `cls` as the first parameter.
- Note that `self` and `cls` are just conventions, could be called anything, and at times it is not clear which to use.

# Class functions and class variables

## A confusing but correct example

```
class Example(object):

    instances = []

    def __init__(self):
        self.id = len(self.instances)
        self.instances.append(self)

a = Example()
b = Example()
print(a.id)
print(b.id)
print(Example.instances)
print(a.instances)
print(b.instances)
print(Example.id)
```

# Inheritance and polymorphism

## The real strength of OOP

- The concept of class can be made much more powerful by the use of sub-classing.
- A sub-class is a class which *inherits* the features of another class, with the possibility of changing some of them or adding further features.
- Often, classes are exemplified with taxonomies, such as Dog being a sub-class of Mammal.
- More interesting for us, is perhaps a Regularized Linear Regression which we will treat as a sub-class of OLSReg.

# OLSReg

## A class definition example

```
class RidgeReg(OLSReg):

    def __init__(self, l):
        super()
        self.l = l

    def train(self, dataset):
        # Code to fit coefficients to
        # dataset using L2 regularization
        # with self.l as multiplier for
        # the ridge
        ...
```

# Inheritance and polymorphism

## The real strength of OOP

- Note that `__init__` calls `super().__init__()`. This runs the constructor for the super-class of RidgeReg, i.e. OLSReg, and makes sure `coeff` is initialized.
- Other than `__init__` and `train`, we don't implement any other functions.
- The functions `coefficients` and `predict` are both inherited from OLSReg.
- As they only depend on the value of `coeff`, not on how that value was computed (i.e. by `train`), they work just as well for RidgeReg as for OLSReg.