UNIVERSITY OF SAN FRANCISCO

Master of Science in Analytics

Object Oriented Programming

! (Natural Language Processing)

Object Oriented Programming

- OOP is a framework for building programs / applications / software
- Elements of object oriented languages:
  - Encapsulation
  - Inheritance
  - Polymorphism
- In Python (also: Java, C++, ...), you can:

Do this	**... for example:**
Create an object	`my_file = open("my_file.txt", "wb")`
Use an object	`my_file.write("I want to learn OOP.")` `my_file.close()`
Destroy an object	`my_file = None`

Example: person

- What makes a person?
  - Name (first names, surname)
  - DOB
  - Biography
  - [...]
- What can a person do?
  - Get married
  - Add to biography
  - Change names
  - Print to a file
  - [...]

- Example fields:
  - Guido van Rossum
  - 1956-01-31
  - Created python
- Example actions:
  - Married Kim Knapp
  - Biography additions: Working for Dropbox (2012)

Python Class: self, __init__

- The class keyword
  - Defines a class
  - Next token (person) names the class
  - All code in class indented
- Encapsulation
  - Fields and actions contained within the class
  - Actions: called “methods” (or “behaviours” or…)
- Keywords ⇒ key concepts
  - self: a particular *instance* of an object; required as first argument to each method
  - __init__: automatically called when the object is instantiated; also defines the class’ fields

```
class person:
    '''
    Comments for a person class!
    '''

    def __init__(self, name):
        '''
        First function called.
        That's 2 underscores before
        and 2 underscores after.
        '''
        self.first_names = name['f']
        self.surname = name['last']
        self.biography = []
        self.spouse = None

    def add_to_bio(self, words):
        '''
        A function to add to bio.
        '''
        self.biography.append(words)
```

Declaration of person

```
class person:

    def __init__(self, name):
        self.first_names = name['f']
        self.surname = name['last']
        self.biography = []
        self.spouse = None

    def add_to_bio(self, words):
        self.biography.append(words)

    def change_name(self, name):
        pass

    def change_spouse(self, spouse):
        pass
```

- Code needed to set up example

```
name = dict()
name['f'] = 'Guido'
name['last'] = 'van Rossum'
```

- Need an instance? Assign to a variable from a class' name

```
p = person(name)
```

- Using an instance? Use a "."

```
p.add_to_bio('2013: Dropbox')
```

- Other than the use of "self", code in methods = code in functions

Limitations & Good Practices

- Limitations of python
  - All fields (or methods) are accessible:

```
p = person(name)
p.biography = ['No'] # yuck!
```

  - Violation of proper encapsulation
- Good practices:
  - In each class:
    - Create accessors for fields
    - Create mutators for fields
    - Rely on methods more than on data
    - Create tiny functions
  - Between classes (collaborations):
    - Create specialty classes — eg. for I/O
    - … but be practical

```
class person:

    def __init__(self, name):
        self.name = dict()
        self.biography = []
        self.spouse = None
        self.change_name(name)

    def get_name(self, name):
        pass

    def set_name(self, name):
        pass

    def to_string(self):
        pass
```

Practice

- Create a class “person” so that the following code runs correctly:

```
from person import person
if __name__ == '__main__':
    guido_name = {'f': 'Guido', 'last': 'van Rossum'}
    guido = person(guido_name)

    guido.add_to_bio('1956: Born in the Netherlands')
    guido.add_to_bio('1989: Started writing python')
    guido.add_to_bio('2012: Started working for Dropbox')

    kim_name = {'f': 'Kim', 'last': 'Knapp'}
    kim = person(kim_name)
    guido.change_spouse(kim)

    print guido.to_string()
```

- Output

```
Guido van Rossum
* Married to: Kim Knapp
* Biography:
  + 1956: Born in the Netherlands
  + 1989: Started writing python
  + 2012: Started working for Dropbox
```

- Solution at at GitHub as person.py

Nomenclature

- Class — (“person”) the blueprint / template
- Object — (“guido” or “kim”) an instance of a class
- Instantiation — the act of creating an object from a class
- “On” — (apologies to English L1) the preposition used for a function of a class. For example, “the get_name function on person…”.
- Attributes
- Constructor

Suppose…

- What if we had a student?
  - Attributes
    - All the attributes from person
    - Also: student ID (string?); courses (list?)
  - Methods
    - All the methods from person — except biography includes courses taken
    - Also: accessors/mutators from for student ID; courses
- Goals
  - Define student as a person
  - Where possible, re-use the attributes and methods from person
  - Make changes only for differences
- Inheritance

Code Start

- Some (not very good) python to get started:

```
if __name__ == '__main__':
    s = get_a_person({'f': 'Elle', 'last': 'Woods'}
    s.add_to_bio('2001: Entered Harvard')
    s.add_to_courses('2017: Web Analytics')
    s.add_to_courses('2017: Natural Language Processing')
    print s.to_string()
```

- Sample output

```
Elle Woods
* Biography:
  + 2001: Entered Harvard
  + Took course = 2002: Web Analytics
  + Took course = 2002: NLP
```

Step 1: student is_a person

- Extending from *person*, build a *student*
  - A student has everything a person has (name, bio, spouse)
  - A student also has a student ID (string) and a set of courses taken
- In Python
  - Define a student as a new type of person

```
from person import person  # Assumes person is defined in person.py
class student(person):
```

  - By default, this allows student objects to inherit everything from the person class

Step 2: Copy From person

- Keep the person fields, add new student-specific fields:
  - Use the same __init__(...) signature:

```
class student(person):
    def __init__(self, name):
```

  - In student.__init__(...), call the __init__(...) defined on person (#pro_move):

```
person.__init__(self, name)
```

  - Finally, add student-specific fields:

```
self.courses = []
self.student_id = '00000000'
```

- Add student-specific methods

```
def get_student_id(self):
    return self.student_id
```

Subtleties

- You DO NOT need to re-write any function on student:
  - If that function is defined in person
  - ... and if the function does what you need it to
- Rule of thumb:
  - Need changes to a person function? Re-write it. The SIGNATURE must be identical.
  - You may not know what type (student / person / student) you have — eg.:

```
from person import person
from student import student

person_type = ''
while person_type != 'person' and person_type != 'student':
    person_type = raw_input('Enter person or student: ')
p = None
if person_type == 'person':
    p = person(name)
else:
    p = student(name)
return p
```

get_biography(...) on student

- A person biography has accomplishments (ala Guido)
- A student biography also includes courses taken:

```
def get_biography(self):

    bio = ''

    for bio_entry in self.biography:
        bio += '  + ' + bio_entry + '\n'

    for course in self.courses:
        bio += '  + Took course = ' + course + '\n'

    return bio
```

- This function on student has the same signature as the one on person

Polymorphism

- You may have an object but don't know its type:

```
p = get_a_person_or_student({'f': 'Elle', 'last': 'Woods'})
```

- You may have a function defined twice (once on person; once on student)
  - An OO language will call the function closest to the object you have:
    - Call p.get_biography() on a student object? ⇒ student version
    - Call p.get_biography() on a person object ⇒ person version
    - If a function is not overridden, the person version will be called.

student All Together

```
class student(person):
    def __init__(self, name):
        person.__init__(self, name)    # Get all person fields
        self.courses = []
        self.student_id = '00000000'

    def get_student_id(self):
        return self.student_id

    def set_student_id(self, sid):
        self.student_id = sid

    def get_courses(self):
        return self.courses

    def add_to_courses(self, course):
        self.courses.append(course)

    def get_biography(self):
        bio = ''
        # ... etc.
```

Do This At Home

- What's at GitHub:
  - Classes: person.py, student.py
  - Test (main) functions: test-person.py, test-student.py
- This code is for demonstration purposes only; perfective fixes:
  - display: the code performing I/O should be its own class
  - factory: the code for constructing objects should be its own class
  - The factory should have the "add_to_courses(...)" calls