UNIVERSITY OF
SAN FRANCISCO
CHANGE THE WORLD FROM HERE

Classification and OOP in Python

~ (Machine Learning)

Classification

- Attach a label to an instance
- More formally:
  - Given:
    - A set of *n* training instances (train_x) and a feature matrix representing those instances
    - A corresponding set of *k* training labels (train_y)
  - Infer the most likely label (test_y) for one or more unseen instances (test_x)
- Examples:
  - Decide whether an email is spam
  - Determine a speaker's native language
  - Handwritten digit recognition:

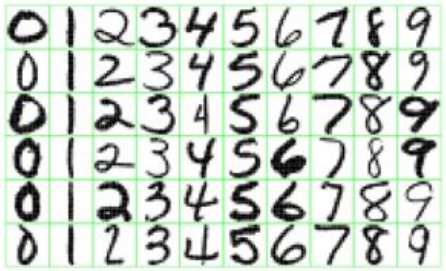

Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

scikit-learn (sklearn)

- General-purpose machine learning toolkit with modules for:
  - Regression — continuous response variable
  - Classification — discrete response variable
  - Clustering — no response variable
- Various algorithms accessible through Python
  - Many modules written in Cython (C-to-Python bridge)
  - Some modules are wrappers around existing libraries (LIBSVM, LIBLINEAR)
- Open source; built on well-known open source projects, including:
  - NumPy
  - SciPy
  - Matplotlib

ML in sklearn

- The code for calling sklearn (regression, classification) follows a pattern:

```
from sklearn.sub_package import algorithm

model = algorithm()
model.fit(train_X, train_Y)
model.predict(test_X)
```

  - ... but “algorithm” (and the sub_package from which it comes) need real values
- Inexperienced ML practitioners might:
  - Run all algorithms vs. data
  - Pick the best algorithm
- We will:
  - Learn how the algorithms work and pick the best one(s) for the data
  - Learn how to handle data properly so algorithms work in many cases

OOP in Python

- OOP is the framework for many python applications, including sklearn
- All OOP elements exist in python:
  - **Encapsulation** (variables and functions in one place) — but everything is `"public"`
  - **Inheritance** (classes get variables and functions from their parents)
  - **Polymorphism** (a subclass can act as a member of its superclass)
- In Python, you can:

Do this	**... for example (in Python):**
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 (indented) within the class
  - Methods / functions also contained within the class, also indented
- 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):
        '''
        Constructor.
        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('2012: Dropbox')
```

  - NOTE: all fields are public!
- Other than the use of "self", code in class functions is no different from global functions

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 (special) class of person

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

  - By default, student inherits everything from person

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
```

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

- As in any OO language, 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.
```

Lab

- Lab 1 on Canvas
- Complete two implementations of logger class:
  - 1: stdout_logger to output log to console
  - 2: file_logger to output log to a file