Tuples and classes

Student management system

- Each student has a name, a major, and a list of grades
- We want to be able to easily access data about students, construct lists of them, etc

Student management program
First attempt: lists and dicts

```
student_majors = {
    "Jaromir": "compsci",
    "Katka": "history",
    "Zoltan": "poetry",
    "Jan": "compsci",
    "Bob": "nameless dread",
    "Sally": "creative exhaustion",
    "Vladimir": "spatula science",
    "Ilya": "archeology",
    "Ulrich": "compsci",
}

student_grades = {
    "Jaromir": [100, 80, 70, 32],
    "Katka": [100, 90, 70, 32],
    "Zoltan": [100, 85, 72, 92],
    "Jan": [99, 80, 70, 62],
    "Bob": [100, 80, 70, 80],
    "Sally": [100, 80, 73, 32],
    "Vladimir": [100, 50, 80, 50],
    "Ilya": [95, 70, 70, 30],
    "Ulrich": [100, 80, 70, 40],
}
```

```
def student_average_grade(student_name):
    """signature: str -> float
    return the average grade for the named student"""
    return (sum(student_grades[student_name]) /
      len(student_grades[student_name]))

def add_new_student(student_name, major):
    """signature: str, str, str -> NoneType
    Create a new student, of the give name,
    with the given major"""
    student_majors[section_name] = major
    student_grades[student_name] = []

def add_grade(student_name, grade):
   """signature: str, float -> NoneType
   Add a grade to the student's record"""
   student_grades[student_name].append(grade)
```

```python
def get_best_student_name():
    """signature: () -> str
    return the name of the student with the best average"""
    best_name = ""
    best_grade = 0.0
    for name in student_grades.keys():
        avg = student_average_grade(name)
        if avg > best_grade:
            best_grade = avg
            best_name = name
    return best_name

def get_best_student_grade():
    """signature: () -> float
    return the grade of the student with the best average"""
    best_name = ""
    best_grade = 0.0
    for name in student_grades.keys():
        avg = student_average_grade(name)
        if avg > best_grade:
            best_grade = avg
            best_name = name
    return best_grade
```

Tuples: an introduction

```
def get_best_student():
    """signature: () -> tuple(float, str)
    return the name grade of the best student"""
    best_name = ""
    best_grade = 0.0
    for name in student_grades.keys():
        avg = student_average_grade(name)
        if avg > best_grade:
            best_grade = avg
            best_name = name
    return (best_grade, best_name)
```

Create a tuple

```
# Put an x,y coordinate pair into a tuple
>>> position = (34, 90)

# Elements of a tuple need not be of the same type
>>> name = "Bartholomew"
>>> age = 89
>>> gpa = 3.4
>>> person = (name, age, gpa)

# Yep, it's a tuple
>>> type(person)
<class 'tuple'>
```

Examining a tuple

```
>>> position
(34, 90)
# Assign new variables from the tuple's content
>>> (xpos, ypos) = position
>>> xpos
34
>>> ypos
90
# Or access its values by position, like a list
>>> position[0]
34

>>> person
('Bartholomew', 89, 3.4)
# Position access
>>> person[2]
# Destructuring assignment
>>> (n, a, g) = person
>>> n
'Bartholomew'
```

Iteration with tuples

```
i = 0
for char in "hello":
    print("Char "+char+" is at position "+str(i))
    i+=1
```

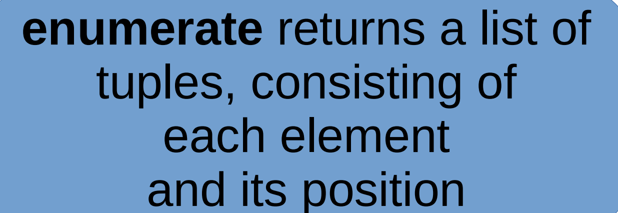
```
--------
```

```
Char h is at position 0
Char e is at position 1
Char l is at position 2
Char l is at position 3
Char o is at position 4
```

Iteration with tuples

```
for (i, char) in enumerate("hello"):
   print("Char "+char+" is at position "+str(i))
```

```
Char h is at position 0
Char e is at position 1
Char l is at position 2
Char l is at position 3
Char o is at position 4
```




```
>>> list(enumerate("hello"))
[(0, 'h'), (1, 'e'), (2, 'l'), (3, 'l'), (4, 'o')]
```

Student management program
Second attempt: tuples

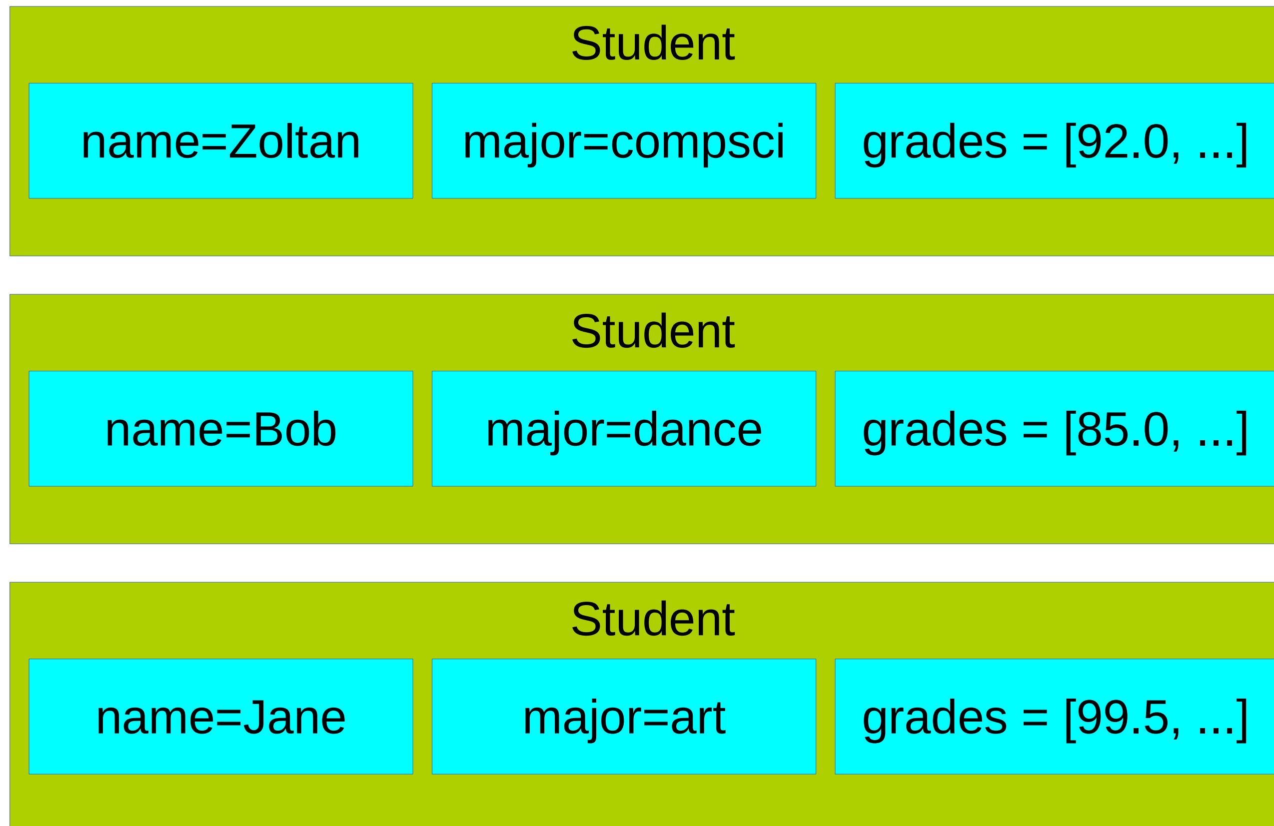
```
student_grades = {
    "Jaromir": ("compsci", [100, 80, 70, 32]),
    "Katka": ("histry", [100, 90, 70, 32]),
    "Zoltan": ("poetry", [100, 85, 72, 92]),
    "Jan": ("compsci", [99, 80, 70, 62]),
    "Bob": ("nameless dread", [100, 80, 70, 80]),
    "Sally": ("creative exhaustion", [100, 80, 73, 32]),
    "Vladimir": ("spatula science", [100, 50, 80, 50]),
    "Ilya": ("archeology", [95, 70, 70, 30]),
    "Ulrich": ("compsci", [100, 80, 70, 40]),
}
```

```
def student_average_grade(student_name):
    """signature: str -> float
    return the average grade of the given student"""
    (major, grades) = student_grades[student_name]
    return sum(grades) / len(grades)
```

```
def add_student_grade(student_name, grade):
    """signature: str, float -> NoneType
    Add the grade to the student's record"""
    (major, grades)  = student_grades[student_name]
    grades.append(grade)

def add_new_student(student_name, major):
    """signature: str, str, str -> NoneType
    Create a new student, of the give name,
    with the given major"""
    student_grades[student_name] = (major, [])

def add_grade(student_name, grade):
   """signature: str, float -> NoneType
   Add a grade to the student's record"""
   (major, grades)  = student_grades[student_name]
   grades.append(grade)
```

A **Student** value

```
# Create two new students
>>> bob = Student("Bob", "dance")
>>> jane = Student("Jane", "philosophy")

# What types is it?
>>> type(bob)
<class '__main__.Student'>

# Access a property of the student
>>> bob.major
'dance'

# the other student's properties can differ
>>> jane.major
'philosophy'

# Give student some grades
>>> bob.add_grade(95.0)
>>> bob.add_grade(75.0)

# And then examine what we know
>>> bob.average_grade()
85.0
```



Each **Student** value contains certain *properties:* a name, a major, and a list of grades. The set of properties are the same for all students, but can differ in value. For example:

```
>>> zoltan.major
'compsci'
>>> jane.major
'art'
>>> bob.major
'dance'
```

type	examples of values
`int`	`3` `-45`
`float`	`9.0` `2323.2`
`str`	`"a spatula"` `"99.3"`
`list`	`[]` `["hello", "argonaut"]`
`Student`	`Student("Bob", "dance")`

Some new vocab

`bob = Student("bob", "dance")`

- "class"
  - a type defined by the programmer, rather than built-in to the language
  - for example, `Student`
- "object" or "instance"
  - a value whose type is a class
  - for example, `bob`
- Just as `3` is a value of type `int`, Bob is a value of type `Student`
- "method"
  - a function defined as part of a class
- "property"
  - a variable defined as part of a class

Designing the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.83333333333333
>>>
```

Designing the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.83333333333333
>>>
```

Every student has a **grades** property. It's a list of floats.

Properties are accessed using the dot notation you've used before. The object on the left, then a period, then the name of the property. You can use them like normal variables (e.g. read them, assign them, etc), but each student has their own copy.

Designing the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.83333333333333
>>>
```

Designing the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.83333333333333
>>>
```

And a **major** property, also a str

Designing the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.83333333333333
>>>
```

Every student has an **add_grade** method for adding a grade to their records

Designing the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.83333333333333
>>>
```

And a similar **add_grades** method for multiple grades at once

Designing the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.83333333333333
>>>
```

Every student has an **average_grade** method for calculating their average

Designing the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.8333333333333
>>>
```

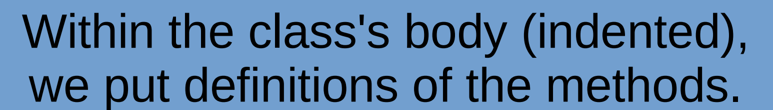
Finally, we have a way to create a new Student object: a *constructor*

Creating a new class

```
class Student:
```

Creating a new class

```
class Student:
    def add_grade(self, grade):
        self.grades.append(grade)
```



Creating a new class

```
class Student:
    def add_grade(self, grade):
        self.grades.append(grade)
```

Every method takes an implicit first parameter which must be named **self**. This value refers to *the present object*, in this case the student we're accessing. Through the special **self** parameter, we can access the object's properties, such as **grades**.

Creating a new class

```
class Student:
    def add_grade(self, grade):
        self.grades.append(grade)
    def add_grades(self, grades):
        for grade in grades:
            self.add_grade(grade)
    def average_grade(self):
        grade_sum = 0
        for grade in self.grades:
            grade_sum += grade
        return grade_sum / len(self.grades)
```

Add more methods. Every method has the **self** parameter.

Creating a new class

```
class Student:
    def __init__(self, name, major):
        self.name = name
        self.major = major
        self.grades = []
    def add_grade(self, grade):
        self.grades.append(grade)
    def add_grades(self, grades):
        for grade in grades:
            self.add_grade(grade)
    def average_grade(self):
        grade_sum = 0
        for grade in self.grades:
            grade_sum += grade
        return grade_sum / len(self.grades)
```

Finally, create a special method named **__init__**. It's the *constructor*. Its job is to initialize all the object's properties. It's called when we create a new Student object, and its parameters

Using the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.8333333333333
>>>
```

Calls the constructor in the Student class.

```
def __init__(self, name, major):
    ...
```

Using the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.83333333333333
>>>
```

Calls the **average_grade** method:

```
def average_grade(self):
    ...
```

Using the Student class

```
>>> bob = Student("Bob", "dance")
>>> bob.add_grade(99.0)
>>> bob.add_grades([95.0, 90.5])
>>> bob.grades
[99.0, 95.0, 90.5]
>>> bob.name
'Bob'
>>> bob.major
'dance'
>>> bob.average_grade()
94.83333333333333
>>>
```

Access the **major** property, which was created by the constructor

Using the Student class

```
students = [
    Student("Jaromir", "compsci"),
    Student("Katka", "histry"),
    Student("Zoltan", "poetry"),
    Student("Jan", "compsci"),
    Student("Bob", "nameless dread"),
    Student("Sally", "creative exhaustion"),
    Student("Vladimir", "spatula science"),
    Student("Ilya", "archeology"),
    Student("Ulrich", "compsci"),
 ]

students[0].add_grades([100, 80, 70, 32])
students[1].add_grades([100, 90, 70, 32])
students[2].add_grades([100, 85, 72, 92])
students[3].add_grades([0, 80, 70, 0])
students[4].add_grades([100, 80, 70, 80])
students[5].add_grades([100, 80, 73, 32])
students[6].add_grades([100, 50, 80, 50])
students[7].add_grades([95, 70, 70, 30])
students[8].add_grades([0, 80, 70, 0])
```

Using the Student class

```
def find_failing_students():
    """signature: () -> list(Student)
    Return all students with failing averages"""
    found = []
    for student in students:
        if student.average_grade() < 60:
            found.append(student)
    return found

>>> find_failing_students()
[<__main__.Student object at 0x7f6495089b38>, <__main__.Student
object at 0x7f6495089c50>]
>>> find_failing_students()[0].name
'Jan'
>>> find_failing_students()[0].major
'compsci'
>>> find_failing_students()[0].average_grade()
37.5
>>>
```

Student management program
Third attempt: classes and objects

School

Section

title="comp112-0"

instructor="Morehouse"

students

Student

name=Zoltan

major=compsci

grades = [92.0, ...]

Student

name=Katka

major=poetry

grades = [94.5, ...]

Section

title="comp112-1"

instructor="Epstein"

students

Student

name=Bob

major=drawing

grades = [89.0, ...]

```
class Student:
    def __init__(self, name, major):
        """signature: Student, str, str -> NoneType
        Initialize the present student"""
        self.name = name
        self.major = major
        self.grades = []
    def add_grade(self, grade):
        """signature: Student, float -> NoneType
        Add a grade to the student's record"""
        self.grades.append(grade)
    def add_grades(self, grades):
        """signature: Student, list(float) -> NoneType
        Add multiple grades"""
        for grade in grades:
            self.add_grade(grade)
    def average_grade(self):
        """signature: Student -> float
        Get the average of this student's grades"""
        grade_sum = 0
        for grade in self.grades:
            grade_sum += grade
        return grade_sum / len(self.grades)
```

```
class Section:
    def __init__(self, title, instructor):
        """signature: Section, str, str -> NoneType
        initialize the present section"""
        self.title = title
        self.instructor = instructor
        self.students = []
    def add_student(self, student):
        """signature: Section, str -> NoneType
        Add a new student of the given name"""
        self.students.append(student)
    def get_best_student(self):
        """signature: Section -> tuple(float, str)
        Return the grade and name of the best student
        in this section"""
        best_so_far = (0, "")
        for student in self.students:
            student_name = student.name
            student_grade = student.average_grade()
            if student_grade > best_so_far[0]:
                best_so_far = (student_grade, student_name)
        return best_so_far
```

Continued on next slide....

```
class Section:
    # Continued from previous slide
    def average_grade(self):
        """signature: Section -> float
        return the average grade of all students in this section""
        grade_sum = 0
        for student in self.students:
            grade_sum += student.average_grade()
        return grade_sum / len(self.students)
    def find_student(self, student_name):
        """signature: Section, str -> Student
        Return a student of the given name"""
        for student in self.students:
            if student.name == student_name:
                return student
        raise ValueError("Can't find desired student")
```

```
class School:
    def __init__(self):
        """signature: School -> NoneType
        Initialize an empty School, with no sections"""
        self.sections = []
    def add_section(self, section):
        """signature: School, Section -> NoneType
        Add a section to the School"""
        self.sections.append(section)
    def add_student(self, section_name, student_name, major):
        """signature: School, str, str, str -> NoneType
        add a student wit the given name and major to the
        already-existing section"""
        section = self.find_section(section_name)
        new_student = Student(student_name, major)
        section.add_student(new_student)
    def find_section(self, section_name):
        """signature: School, str -> Student
        return a section of the given name"""
        for section in self.sections:
            if section.title == section_name:
                return section
        raise ValueError("Can't find desired section")
    def find_student(self, student_name):
        """signature: School, str -> Student
        return a section of the given name"""
        for section in self.sections:
            try:
                return section.find_student(student_name)
            except ValueError:
                pass
        raise ValueError("Can't find desired student")
```

```
school = School()
school.add_section(Section("comp112-0", "Morehouse"))
school.add_section(Section("comp112-1", "Epstein"))
school.add_section(Section("comp112-2", "Thayer"))
school.add_student("comp112-0", "Jaromir", "compsci")
school.add_student("comp112-0", "Katka", "history")
school.add_student("comp112-0", "Zoltan", "poetry")
school.add_student("comp112-1", "Jan", "compsci")
school.add_student("comp112-1", "Bob", "nameless dread")
school.add_student("comp112-1", "Sally", "creative exhauastion")
school.add_student("comp112-2", "Vladimir", "spatula science")
school.add_student("comp112-2", "Ilya", "archeology")
school.add_student("comp112-2", "Ulrich", "compsci")
school.find_student("Jaromir").add_grades([100, 80, 70, 32])
school.find_student("Katka").add_grades([100, 90, 70, 32])
school.find_student("Zoltan").add_grades([100, 85, 72, 92])
school.find_student("Jan").add_grades([99, 80, 70, 62])
school.find_student("Bob").add_grades([100, 80, 70, 80])
school.find_student("Sally").add_grades([100, 80, 73, 32])
school.find_student("Vladimir").add_grades([100, 50, 80, 50])
school.find_student("Ilya").add_grades([95, 70, 70, 30])
school.find_student("Ulrich").add_grades([100, 80, 70, 40])
```

```
>>> school.find_section("comp112-2").get_best_student()
(72.5, 'Ulrich')
>>> school.find_student("Jan")
<__main__.Student object at 0x7faa4a572048>
>>> school.find_student("Jan").major
'compsci'
>>> school.find_student("Jan").average_grade()
77.75
>>> school.find_student("Jan").grades[3]
62
>>> school.find_section("comp112-0").average_grade()
76.91666666666667
>>> school.find_section("comp112-1").instructor
'Epstein'
>>>
```