Advanced Programming

Coordinator and Examiner

- Mats Dahllöf

Teachers

- Artur Kulmizev
- Per Starbäck

TAs

- Gustav Finnveden
- Harm Lameris
- Xingran Zhu (朱星燃, Winstead)

Learning outcomes

- Use and motivate object-oriented design in programming
- Solve programming tasks of relevance to language technology, including processing of text data collections and application of numerical methods;
- implement algorithms for the following tasks and analyse their time complexity: standard operations on stacks, queues, and lists, search, and sorting

Modules

- Object-oriented programming
- Algorithms and complexity
- Programming for LT applications
- Object-oriented design
- Q&A
- Written test

Examination and grading

Four assignment packages and a final written test

To pass the course (grade *G*) you need to pass all assignments, and the written exam.

To pass with distinction (*VG*) you also need to achieve that result at the written exam.

Assignments

There are four *assignment packages*.

You are “strongly encouraged to work with and complete the assignments according to the pace of the course.”

The hard deadlines are not until after the written exam though. See info in Studium for details!

Most assignments will be done in assigned pairs, different each time

Object-oriented programming

Object-oriented programming

- You will learn to define and use objects
- Sometimes other additional Python parts not covered in P1
- There will be a mandatory assignment (next week)
- But also you'll use what you learn now in later parts of the course

Python keywords

These are all the *keywords* in Python:

```
False None True and as assert break class
continue def del elif else except finally
for from global if import in is lambda
nonlocal not or pass raise return try
while with yield
```

(You can get this list with `import keyword` and `keyword.kwlist`.
This is from Python 3.6 – the list will be slightly different depending on version.)

Python keywords

These are all the *keywords* in Python:

```
False None True and as assert break class
continue def del elif else except finally
for from global if import in is lambda
nonlocal not or pass raise return try
while with yield
```

(You can get this list with `import keyword` and `keyword.kwlist`.
This is from Python 3.6 – the list will be slightly different depending on version.)

Python keywords

These are all the *keywords* in Python:

```
False None True and as assert break class
continue def del elif else except finally
for from global if import in is lambda
nonlocal not or pass raise return try
while with yield
```

(You can get this list with `import keyword` and `keyword.kwlist`.
This is from Python 3.6 – the list will be slightly different depending on version.)

Python keywords

These are all the *keywords* in Python:

```
False None True and as assert break class
continue def del elif else except finally
for from global if import in is lambda
nonlocal not or pass raise return try
while with yield
```

Python keywords

These are all the *keywords* in Python:

```
False None True and as assert break class
continue def del elif else except finally
for from global if import in is lambda
nonlocal not or pass raise return try
while with yield
```

The simplest Python statement of them all. It does nothing!

Using `pass`

```
def contemplate_the_universe():
    pass
```

Types

We have used many different standard types in P1:

Types

We have used many different standard types in P1:

- int
- float
- str
- list
- tuple
- dict
- function
- bool
- range
- NoneType
- ...

And more in various modules

```
>>> from nltk.corpus import gutenberg
>>> emma = gutenberg.sents('austen-emma.txt')
>>> len(emma)
7752
>>> emma[5333]
['Supper', 'was', 'announced', '.']
>>>
```

And more in various modules

```
>>> from nltk.corpus import gutenberg
>>> emma = gutenberg.sents('austen-emma.txt')
>>> len(emma)
7752
>>> emma[5333]
['Supper', 'was', 'announced', '.']
>>> type(emma)
<class 'nltk.corpus.reader.util.StreamBackedCorpusView'>
>>>
```

Methods

One way these types differ is in which methods they have. In P1 we used several methods on lists and strings, some returning values and some not.

```
>>> words = ['one', 'two', 'three']
>>> words[1].upper()
'TWO'
>>> words.append('four')
>>> ' ~ '.join(words)
'one ~ two ~ three ~ four'
>>> words.sort()
>>> words
['four', 'one', 'three', 'two']
>>>
```

Functions vs Methods

```
animal = 'python'
```

Functions vs Methods

```
animal = 'python'
```

What is the length of that string?

```
len(animal)
```
$\longmapsto$ 6

Functions vs Methods

```
animal = 'python'
```

What is the length of that string?

`len(animal)` $\longmapsto$ 6

What is the uppercase version of that string?

`animal.upper()` $\longmapsto$ 'PYTHON'

Functions vs Methods

```
animal = 'python'
```

What is the length of that string?

`len(animal)` $\longmapsto$ 6

Start with function, give it a string

What is the uppercase version of that string?

`animal.upper()` $\longmapsto$ 'PYTHON'

Start with string, give it a method

Functions vs Methods

```
animal = 'python'
```

What is the length of that string?

`len(animal)` $\longmapsto$ 6

Start with function, give it a string

What is the uppercase version of that string?

`animal.upper()` $\longmapsto$ 'PYTHON'

Start with string, give it a method

The latter way of doing things is more object-oriented.

Object-oriented programming

From Wikipedia:

> programming languages; see List of object-oriented progr
>
> **Object-oriented programming** (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as *attributes* or *properties*), and code, in the form of procedures (often known as *methods*).
>
> A feature of objects is that an object's own procedures

Object-oriented programming

From Wikipedia:

> programming languages; see List of object-oriented progr
>
> **Object-oriented programming** (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as *attributes* or *properties*), and code, in the form of procedures (often known as *methods*).
>
> A feature of objects is that an object's own procedures

When doing `'hello'.upper()` the code for upper-casing is a method that is part of the string object itself.

Example: Turtles

Example: Turtles

There is also a video demo on Studium.

Example: Circles

Example: Circles

(There is also a video at Studium.)

Class

When defining new types we define them with `class`.

Class

When defining new types we define them with `class`.

```
class Book:
    pass
```

Classes have names beginning with a capital letter.

Class

When defining new types we define them with `class`.

```
class Book:
    pass

wd = Book()
wd.title = 'Watership Down'
wd.year = 1972
```

Classes have names beginning with a capital letter.

An instance of the class is created.

Attributes are set.

Class

When defining new types we define them with `class`.

```
class Book:
    pass

wd = Book()
wd.title = 'Watership Down'
wd.year = 1972

drac = Book()
drac.title = 'Dracula'
drac.year = 1897
```

Classes have names beginning with a capital letter.

An instance of the class is created.

Attributes are set.

Class

When defining new types we define them with `class`.

```
class Book:
    pass

wd = Book()
wd.title = 'Watership Down'
wd.year = 1972

drac = Book()
drac.title = 'Dracula'
drac.year = 1897

books = [wd, drac]
for b in books:
    if b.year < 1900:
        print(b.title)
```

Classes have names beginning with a capital letter.

An instance of the class is created.

Attributes are set.

Attributes are used.

```
class Book:
    pass

b = Book()
```

Note that this definition says nothing about what a book is, or what attributes it can have.

```
class Book:
    pass

b = Book()
```

Note that this definition says nothing about what a book is, or what attributes it can have.

If a book always should have certain attributes we want to specify them already when creating the object, as arguments to `Book`.

```
e = Book('Emma', 1815)
```

Initializing an object

That is done with `__init__`. Note the special name with two underscores first and last!

```
class Book:
    def __init__(self, title, year):
        self.title = title
        self.year = year

e = Book('Emma', 1815)
```

Initializing an object

That is done with `__init__`. Note the special name with two underscores first and last!

```
class Book:
    def __init__(self, title, year):
        self.title = title
        self.year = year

e = Book('Emma', 1815)

x = Book()
Traceback (most recent call last):
...
TypeError: __init__() missing 2 required positional
arguments: 'title' and 'year'
```

Note about `self`

```
class Book:
    def __init__(self, title, year):
        self.title = title
        self.year = year

e = Book('Emma', 1815)
```

We call `Book` here with two arguments. In the `def` for `__init__` there are three arguments. The first one is always called `self` and will be the object itself. Then comes the given arguments.

Defining methods

We define new methods with `def` inside a `class`. The first argument is always `self` which will be the object itself.

```
class Book:
    ...

    def is_antique(self):
        return self.year <= 1921
```

Defining methods

We define new methods with `def` inside a `class`. The first argument is always `self` which will be the object itself.

```
class Book:
    ...

    def is_antique(self):
        return self.year <= 1921
```

```
>>> e.is_antique()
True
>>>
```

Defining methods

```
class Book:
    ...

    def is_good(self):
        goodness = 0
        # Needs to fulfill at least two of these
        if self.year > 1950:
            goodness += 1
        if len(self.title) <= 20:
            goodness += 1
        if 'dragon' in self.title.lower():
            goodness += 1
        return goodness >= 2
```

Defining methods

```
class Book:
    ...

    def is_good(self):
        ...
        return goodness >= 2

e = Book('Emma', 1815)
e.is_good()
```

Note that the method's first parameter (`self`) isn't given inside the `()` when the method is invoked.

```
class Book:
    ...

    def year_diff(self, other_book):
        return abs(self.year - other_book.year)

>>> wd.year_diff(drac)
75
>>> drac.year_diff(wd)
75
>>>
```

Name conventions

Names of functions, variables and methods should be lowercase, with words separated by underscores as necessary to improve readability. `n`, `animal`, `largest_planet`, `ideal_sentence_length`, `char_freqs`.

Names of classes should use "CapitalizedWords" with no separation between words if there are multiple words: `Person`, `Planet`, `FarmAnimal`, `MusicAlbum`, `DependencyGraph`.

> *These are the normal naming conventions in Python. See PEP 8 for more. (Conventions are different in other languages.)*

Circles and balls

I changed the name into `balls.py`.

Circles and balls

I changed the name into `balls.py`.

- The balls are objects! These have x **and** y values

Circles and balls

I changed the name into `balls.py`.

- The balls are objects! These have x **and** y values
- They can't go downwards outside the window!

Circles and balls

I changed the name into `balls.py`.

- The balls are objects! These have x **and** y values
- They can't go downwards outside the window!
- Some values are defined as constant "variables", so for example `BALL_COLOR` instead of `(0, 0, 0)`

Circles and balls

I changed the name into `balls.py`.

- The balls are objects! These have x **and** y values
- They can't go downwards outside the window!
- Some values are defined as constant "variables", so for example `BALL_COLOR` instead of `(0, 0, 0)`
- Pressing Esc exits the program

Circles and balls

I changed the name into `balls.py`.

- The balls are objects! These have x **and** y values
- They can't go downwards outside the window!
- Some values are defined as constant "variables", so for example `BALL_COLOR` instead of `(0, 0, 0)`
- Pressing Esc exits the program
- But wait, there's **more**!

Circles and balls

I changed the name into `balls.py`.

- The balls are objects! These have x **and** y values
- They can't go downwards outside the window!
- Some values are defined as constant "variables", so for example `BALL_COLOR` instead of `(0, 0, 0)`
- Pressing Esc exits the program
- But wait, there's **more**!
- (no, actually that's all)

Circles and balls

I changed the name into `balls.py`.

- The balls are objects! These have x **and** y values
- They can't go downwards outside the window!
- Some values are defined as constant "variables", so for example `BALL_COLOR` instead of `(0, 0, 0)`
- Pressing Esc exits the program
- But wait, there's **more**!
- (no, actually that's all)

Circles and balls

I changed the name into `balls.py`.

- The balls are objects! These have x **and** y values
- They can't go downwards outside the window!
- Some values are defined as constant "variables", so for example `BALL_COLOR` instead of `(0, 0, 0)`
- Pressing Esc exits the program
- But wait, there's **more**!
- (no, actually that's all)

In today's exercise you will improve on that.
See "Balls as objects" in this module in Studium!

The future

- I will leave this Zoom open, so you can continue communicating here if you want.
- Tomorrow (1 PM) you can continue to work on this, with help from TAs
- Don't forget there are videos with demos for two parts of this lecture which you can watch for repetition