CSCI2202 Lecture 5: Object-Oriented Programming

TAs: Ehsan Baratnezhad (ethan.b@dal.ca); Precious Osadebamwen (precious.osadebamwen@dal.ca)

Overview

- Python built around objects
- Classes as definitions of object
- Accessing special methods/attributes of objects
- Defining custom classes with custom methods/attributes
- Object oriented programming
- Object hierarchy and inheritance

Every “thing” in python is an object

```
>>> x = 10
>>> type(x)
<class 'int'>
>>> type(5.0)
<class 'int'>
>>> type({})
<class 'dict'>
>>> type([])
<class 'list'>
```

All of these are objects.

Each object is an `instance` of a `class`

Each `class` has

- A definition
- An internal `data representation`
- A set of ways it can be interacted with

In general a `type` defines the interface (interactions) and a `class` defines the entire object.

In modern python `type` and `class` are largely equivalent terms.

`Class` = definition, `object` = instance of class

Class `int:`
Store a +/- whole number
Arithmetic functions (`methods`)
.__add__(x) # x + y
.__eq__(x) # x == y
.__str__() # print(x)

`instantiation`

`x = 5`

`x = int(5)`

Object x`:`
`Class `int``
`-5`
`methods`

```
>>> y = 2
>>> x + y
5
```

```
>>> x.__add__(y)
5
```

```
>>> y = 2
>>> x == y
False
```

```
>>> x.__eq__(y)
False
```

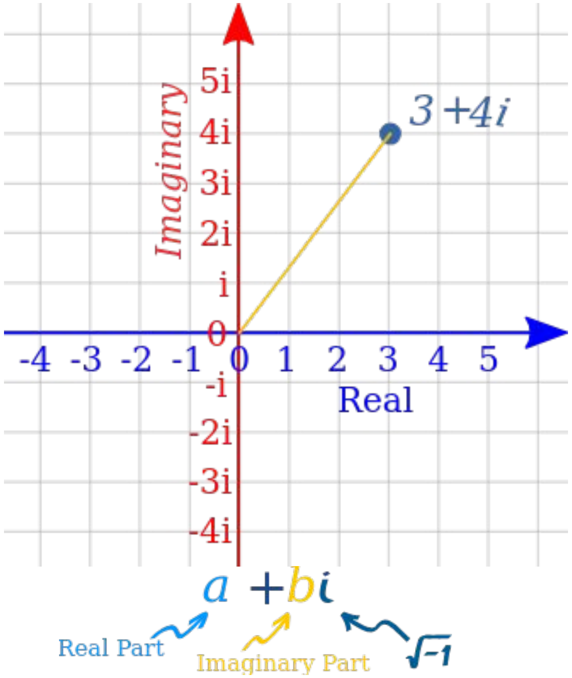
Multiple names can point to same object: `aliasing`

```
>>> x = [1,2,3]
>>> type(x)
<class 'list'>
>>> y = x
>>> id(x) # unique object id
136261838566464
>>> id(y)
136261838566464
```

- Create (“instantiates”) an object defined in the class `list`
- Assign that to the name `x`
- Assign `y` to the same object
- `x` and `y` are references to the same object at the same location in the memory
- This link between x->object and y->object is stored in a `namespace`
- `namespace’s` are also objects (typically an instance of class `dict`)

Defining custom classes

Custom classes can make for simpler code



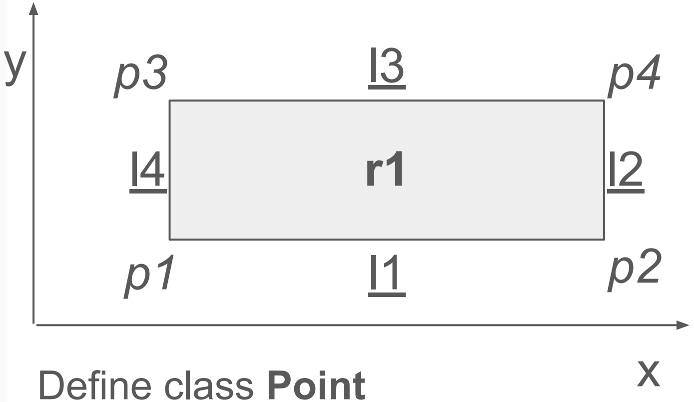
(a+b**i**) + (c+d**i**) = (a+c) + (b+d)**i**

(a+b***i***)(c+d***i***) = ac + ad***i*** + bc***i*** + bd$\boldsymbol{i}^2$

```
x_real_imag = (3, 2) # 3 + 2i

y_real_imag = (1, 7) # 1 + 7i

sum_real = x_real_imag[0] + y_real_imag[0] # 4

sum_imag = x_real_imag[1] + y_real_imag[1] # 9i

added = (sum_real, sum_imag) # 4 + 9i

x = Complex(3, 2) # 3 + 2i

y  = Complex(1, 7) # 1 + 7i

added = x + y # 4 + 9i

multiplied = x * y # -11 + 23i
```

We can combine classes to make more complex objects

```
p1 = Point(1,2)

p2 = Point(6,2)

p3, p4 = Point(1,4),Point(6,4)

l1 = Line(p1, p2)

l2 = Line(p2, p4)

l3, l4 = Line(p3,p4), Line(p1,p4)

r1 = Rectangle([l1, l2,

                l3, l4])
```



Define class **Point**
Define class **Line** using **Point** objects
Define class **Rectangle** using **Line objects**
Can define **functions** (e.g., r1.area() == l1.length() * l2.length())

```
l1.length() # 5
l2.length() # 2
r1.area() # 10
```

Easy to define a new class in python

```python
class MyClass:
    '''Class definitions should have a docstring
    that explains what it does and how to interact
    with it'''
    pass # means python won't crash but class does nothing
```

```python
class CLASSNAME:
    # docstring
    CLASS_BODY
```

- Like functions each class has its own internal `namespace`
- BUT, there are more ways to interact with this `namespace`
- Even basic types like `list` and `dict` in python can be defined as classes under the hood.

Everything in an object is an attribute

```python
class MyClass:
    value1 = 5 # attribute
    value2 = 10 # attribute

    def print_foo():
        # technically an attribute
        # but we typically
        #call class funcs: method
        print('foo')
```

```python
>>> x = MyClass()
>>> x.value1
5
>>> x.value2
10
>>> x.print_foo()
'foo'
```

Default objects are mutable - can change attributes

```python
class MyClass:
    value1 = 5 # attribute
    value2 = 10 # attribute


    def print_foo(): #attribute/method
        print('foo')
```

You CAN modify the **class definition** after defining it but it is like brain surgery on awake person: sometimes needed but high risk and complicated

```python
>>> x = MyClass()
>>> x.value1 = 'bar'
>>> x.value1
'bar'
>>> x.print_baz = lambda: print('baz')
>>> x.print_baz()
'baz'
>>> x = MyClass()
>>> x.value1
5
>>> x.print_baz()
AttributeError: 'MyClass' object has no attribute
'print_baz'
```

Instantiating objects with specific values

__init__ lets us create an object with our own values

```
class MyClass:
    class_val = 'foo'
    def __init__(self, x, y):
        self.value1 = x
        self.value2 = y
x = MyClass('a', 10)
x.value1
'a'
x.value2
10
```

- Class method names that start/end with `__` are called special/magic/dunder methods
- Generally we don't run these directly but they get automatically called when doing certain things
- __init__ automatically gets called like a function when instantiating a class as an object (sometimes called a "constructor")
- Attributes defined during or after __init__ are instance/object attributes, those defined in the class definition itself are class attributes

```
x.class_val
'foo'
```

Be careful with mutable class variables

```
class Dog:

    tricks = []  # mistaken use of a class variable

    def __init__(self, name):

        self.name = name

    def add_trick(self, trick):

        self.tricks.append(trick)

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.add_trick('roll over')

>>> e.add_trick('play dead')

>>> d.tricks # unexpectedly shared by all dogs

['roll over', 'play dead']
```

```
class Dog:

    def __init__(self, name):

        self.name = name

        self.tricks = []  # creates a new empty list for each dog

    def add_trick(self, trick):

        self.tricks.append(trick)

>>> d, e = Dog('Fido'), Dog('Buddy')

>>> d.add_trick('roll over')

>>> e.add_trick('play dead')

>>> d.tricks

['roll over']

>>> e.tricks

['play dead']
```

Think about public vs private attributes

```
class My_Class:
    def set_xy(self, x, y):
        self._x = x
        self._y = y
    def get_sum(self):
        return self._x + self._y
obj = My_Class()
obj.set_xy(3, 5)
print('Sum =', obj.get_sum())
print('_x =', obj._x)
```

- Many OOP languages control whether you can access attributes or methods only from inside an object or externally (public vs private)
- In python everything is always accessible i.e., “public”
- Recommendation in python is to start attributes with underscore, if these are intended to be mostly used locally inside a class, i.e. be considered ”private”
- PEP8: “Use one leading underscore only for non-public methods and instance variables“

You’ve already used many normal and special class methods!

Class methods define interactions (among other things)

Type / class	Objects	Methods (examples)
int	0 -7 42 1234567	.__add__(x), .__eq__(x), .__str__()
str	"" 'abc' '12_ a'	.isdigit(), .lower(), .__len__()
list	[] [1,2,3] ['a', 'b', 'c']	.append(x), .clear(), .__mul__(x)
dict	{'foo' : 42, 'bar' : 5}	.keys(), .get(), .__getitem__(x)
NoneType	None	.__str__()

Example:

The function `str(obj)` calls the methods `obj.__str__()` or `obj.__repr__()`, if `obj.__str__` does not exist.

`print` calls `str`.

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Classes let us organise/package functions for an object

Type / class	Objects	Methods (examples)
int	0 -7 42 1234567	.__add__(x), .__eq__(x), .__str__()
str	"" 'abc' '12_ a'	.isdigit(), .lower(), .__len__()
list	[] [1,2,3] ['a', 'b', 'c']	.append(x), .clear(), .__mul__(x)
dict	{'foo' : 42, 'bar' : 5}	.keys(), .get(), .__getitem__(x)
NoneType	None	.__str__()

```
>>> 'aBCd'.lower()
'abcd'
>>> 'abcde'.__len__()
# .__len__() called by len(...)
5
>>> ['x', 'y'].__mul__(2)
['x', 'y', 'x', 'y']
# eq. to ['x', 'y'] * 2
>>> {'foo' : 42}.__getitem__('foo')
# eq. to {'foo' : 42}['foo']
42
>>> None.__str__()
# used by str(...)
'None'
>>> 'abc'.__str__(), 'abc'.__repr__()
('abc', "'abc'")
```

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

__eq__ and __repr__ are also common special methods

```python
class MyClass:

    def __init__(self, x):

        self.value1 = x

    def __eq__(self, y):

        # all == will be True

        print(f"Ignoring {y}")

        return True

    def __repr__(self):

        print(f"I am {self.value1}")
```

Two other most common special methods are:

- `__eq__` controls how == works with objects of this class
- `__repr__` controls how print (among other things) works with this class

```python
>>> x = MyClass(10)
>>> x == 5
"Ignoring 5"
True
>>> print(x)
"I am 5"
```

Many other “standard” special methods

Function	Special Method Call	Returns
`x == y`	`x.__eq__(y)`	Typically `bool`
`x != y`	`x.__ne__(y)`	Typically `bool`
`<`	`__lt__`	Typically `bool`
`>`	`__gt__`	Typically `bool`
`<=`	`__le__`	Typically `bool`
`>=`	`__ge__`	Typically `bool`
`str(x)`	`x.__str__()`	`str`
`bool(x)`	`x.__bool__()`	`bool`
`int(x)`	`x.__int__()`	`int`

Iterators = object with __iter__ which returns an iterable (object with __next__)

```
L = ['a', 'b', 'c']
it = iter(L) # calls L.__iter__()
next(it) # calls it.__next__()
 'a'
next(it)
 'b'
next(it)
 'c'
next(it)
 StopIteration
```

- Lists are iterable (must support **__iter__**)
- iter returns an iterator (must support **__next__**)
- `next(iterator_object)` returns the next element from the iterator, by calling the iterator_object.__next__(). If no more elements to report, raises exception `StopIteration`
- `next(iterator_object, default)` returns default when no more elements are available (no exception is raised)
- for-loops, comprehensions, map-reduce require iterable objects

Understanding check!

```python
class C:
    def __init__(self, x):
        self.v = x
    def f(self):
        self.v = self.v + 1
        return self.v
```

```python
>>> x = C(10)
>>> print(x.f() + x.f())
?
```

Understanding check!

```python
class C:
    def __init__(self, x):
        self.v = x
    def f(self):
        self.v = self.v + 1
        return self.v
```

```python
>>> x = C(10)
>>> print(x.f() + x.f())
# START: self.v = 10
# EXPRESSION: f() + f()
# run f() -> self.v = 11
# run f() -> self.v = 12
# 11 + 12 = 23
```

More advanced class tricks

Property decorator allows control of attribute changes

```python
class C:
    def __init__(self, in_val):
        self._inside_x = in_val
    @property
    def x(self):
        return (self._inside_x)
    @x.setter
    def x(self, value): # print warnings…
        if type(value) == int:
            self._inside_x = value
    @x.deleter
    def x(self):
        del self._inside_x
```

- Many languages require (or strongly encourage) having special methods for getting or setting attribute values
- Python lets you do this directly but sometimes you may want to add extra logic to control how this is done.
- Easiest way to do this is by using the `@property` decorator

```python
z = C(5)
z.x # getter
z.x = 10 # setter
del z.x # deleter
```

Dataclasses are a convenient way to make data objects

```python
from dataclasses import dataclass

@dataclass

class Student:

    name: str

    major: str

    GPA: float = 0.0
```

- dataclass automates adding useful code for objects designed to store data
- This includes
  - Setting attribute values with specific types
  - Creating default values
  - Comparing data objects __eq__
  - Printing out data objects __repr__
- Can be made immutable `@dataclass(frozen=True)`

PEP8 Style Guide for Classes

- Class names should normally use the CapWords convention.
- Always use self for the first argument to instance methods.
- Use one leading underscore only for non-public methods and instance variables.
- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods (or use @property)
- Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public

Why do we bother with custom classes?

Building your program around classes

Solving problems:

- Top-down design- break big problem into smaller problems and write functions:
  - functional programming where the focus is on functions, lambda's and higher order functions.
  - imperative programming focusing on sequences of statements changing the state of the program

- **OR** Describe the organization of your data and have that reflected in your program:
  - A contact management program will manipulate **Contacts**
  - A drawing program will manipulate a **Canvas**, and perhaps **Lines, Colors**,
  - and **Shapes**
  - Social Media will manipulate **Users**, **Posts**, and **Advertisements**
  - These are the "**nouns**" of these programs
  - We can then define how we interact with these nouns using **verbs** (aka methods/operators)

Object Oriented Programming (OOP)

- OOP is just another programming paradigm
- No single paradigm is the “BEST” each have their roles (lots of modern languages let you mix and match)

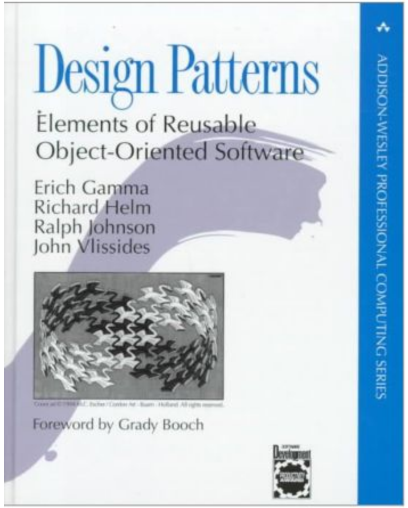
- Core concepts are objects, methods and classes,
  - allowing one to construct abstract data types, i.e. user defined types
  - objects have states (i.e., attributes)
  - methods manipulate objects, defining the interface of the object to the rest of the program’

- OO supported by many programming languages, including Python
- ⅘ most used languages support OOP (Java, C++, Python, C#)

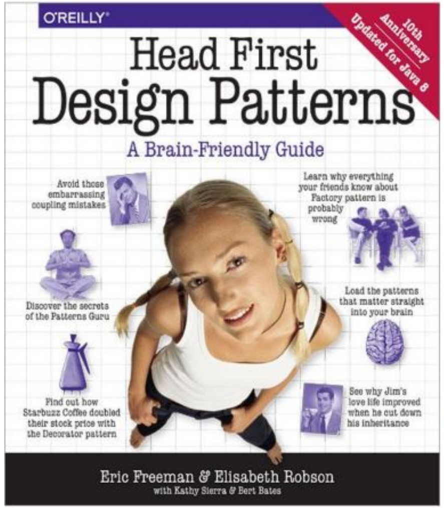
Why is OOP useful?

- OOPs lets us bundle together objects that share:
  - common attributes
  - procedures that operate on those attributes
- Use abstraction to make a distinction between how to Implement an object vs how to use the object
- Create our own classes of objects on top of Python’s basic classes
- Build layers of object abstractions that inherit behaviors/code from other classes of objects
- Easier(?) for lots of developers to work on together

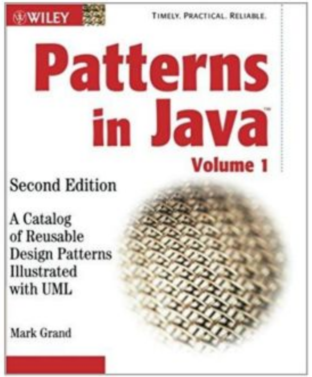
Influential OOP “Design patterns” common in many programs



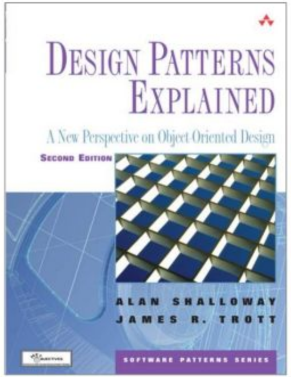
The Classic book 1994
(C++ cookbook)



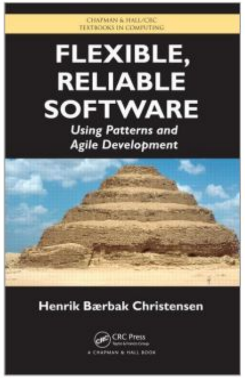
A very alternative book 2004
(Java, very visual)



Java cookbook 2003



Java textbook 2004



Java textbook 2010

Gang of Four

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Let’s dig into OOP a bit more

Student Grades

```
class Assignment:
    def __init__(self, grade):
        if not type(grade) in [int, float]:
            raise ValueError("Not number")
        if not (0 <= grade <= 100):
            raise ValueError("Should be 0-100")
        self.grade = grade
```

```
stu1 = Student("Test Student")
lab1 = Assignment(94)
lab2 = Assignment(50)
stu1.add_grade(lab1)
stu1.add_grade(lab2)
stu1.average_grade()
"Test Student got 72.0"
```

```
class Student:
    def __init__(self, name):
        self.name = name
        self.grades = [ ]
    def add_grade(self, grade):
        if not type(grade) == Assignment:
            raise ValueError
        self.grades.append(grade)
    def average_grade(self):
        vals = [x.grade for x in self.grades]
        mean = sum(vals) / len(vals)
        print(f"{self.name} got {mean}")
        return mean
```

Inheritance is a key concept in OOP

Classes often have overlapping definitions

```
class Person
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
set_name(name)
get_name()

set_address(address)
get_address()

set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

Observation: students and employees are persons with additional attributes

instance

```
Person object
name = 'Mickey Mouse'
address = 'Mouse Street 42, Duckburg'
```

instance

```
Student object
name = 'Donald Duck'
address = 'Duck Steet 13, Duckburg'
id = '1094'
grades = {'programming' : 'A' }
```

```
Employee object
name = 'Goofy'
address = 'Clumsy Road 7, Duckburg'
employer = 'Yarvard University'
```

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Overlapping definitions = duplicated brittle code

```
class Person
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
set_name(name)                 person
get_name()                     attributes

set_address(address)
get_address()

set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

Goal – avoid redefining the 4 methods below from `person` class again in `student` class

person.py

```python
class Person:
    def set_name(self, name):
        self.name = name
    def get_name(self):
        return self.name
    def set_address(self, address):
        self.address = address
    def get_address(self):
        return self.address
```

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Inheritance means we can define shared attributes once

```
class Person
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
set_name(name)
get_name()

set_address(address)
get_address()
```

person attributes

```
set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

class `Student` inherits from class `Person`
class `Person` is the base class of `Student`

person.py

```python
class Student(Person):
    def set_id(self, student_id):
        self.id = student_id
    def get_id(self):
        return self.id
    def set_grade(self, course, grade):
        self.grades[course] = grade
    def get_grades(self):
        return self.grades
```

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Inheritance means we can define shared attributes once

```
class Person
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
set_name(name)                  person
get_name()                      attributes

set_address(address)
get_address()

set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

person.py

```python
class Person:
    def __init__(self):
        self.name = None
        self.address = None
    ...

class Student(Person):
    def __init__(self):
        self.id = None
        self.grades = {}
        Person.__init__(self)
    ...
```

constructor for `Person` class

constructor for `Student` class

Notes

1) If `Student.__init__` is not defined, then `Person.__init__` will be called
2) `Student.__init__` must call `Person.__init__` to initialize the `name` and `address` attributes

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

`super` lets us access the parent/base class

```
class Person
set_name(name)
get_name()

set_address(address)
get_address()
```

```
class Student
set_name(name)
get_name()

set_address(address)
get_address()

set_id(student_id)
get_id()

set_grade(course, grade)
get_grades()
```

person attributes

person.py

```python
class Person:
    def __init__(self):
        self.name = None
        self.address = None
    ...

class Student(Person):
    def __init__(self):
        self.id = None
        self.grades = {}
        Person.__init__(self)
        super().__init__()
    ...
```

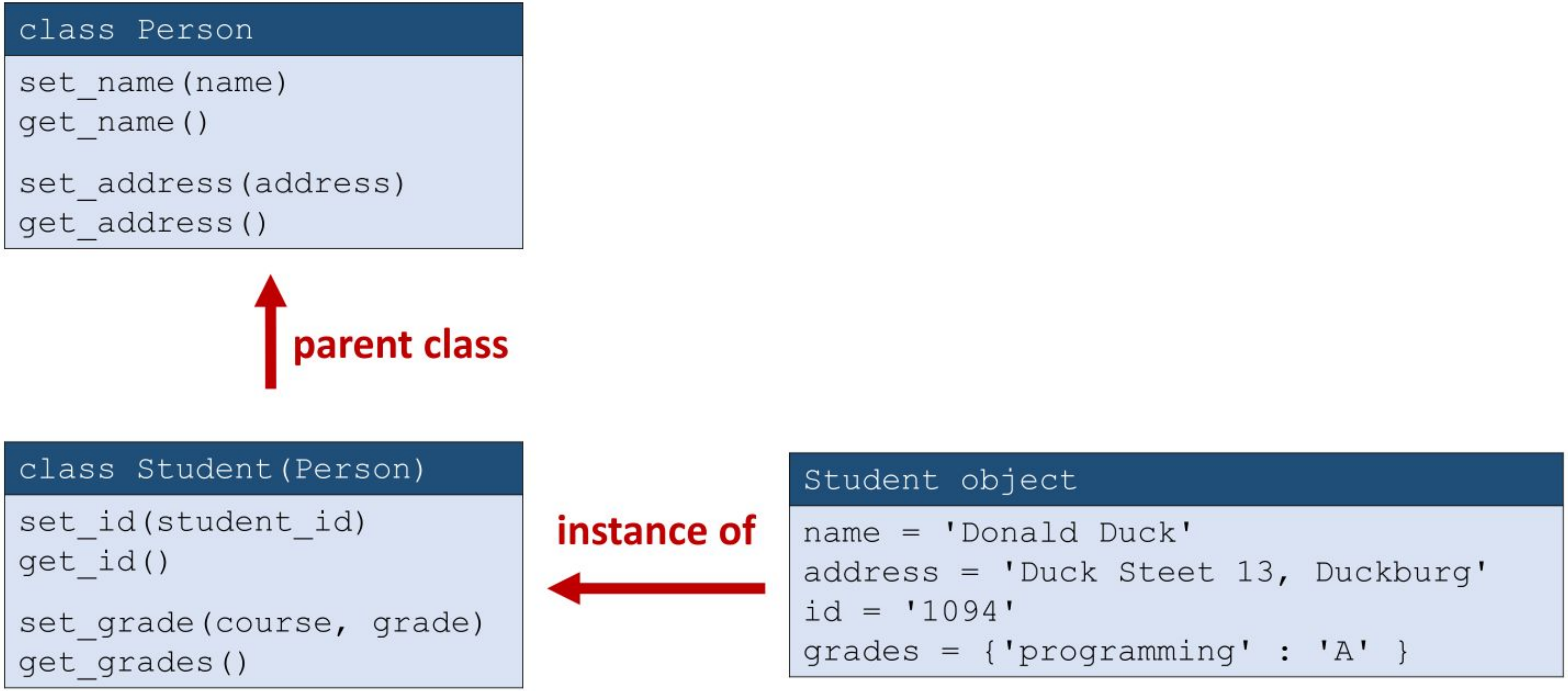
alternative constructor

Notes

1) Function `super()` searches for attributes in base class
2) `super` is often a keyword in other OO languages, like Java and C++
3) Note `super().__init__()` does not need `self` as argument

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

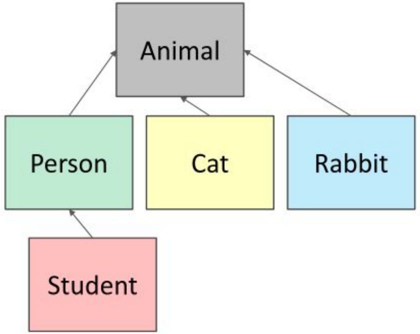
Classes often exist in these types of hierarchies

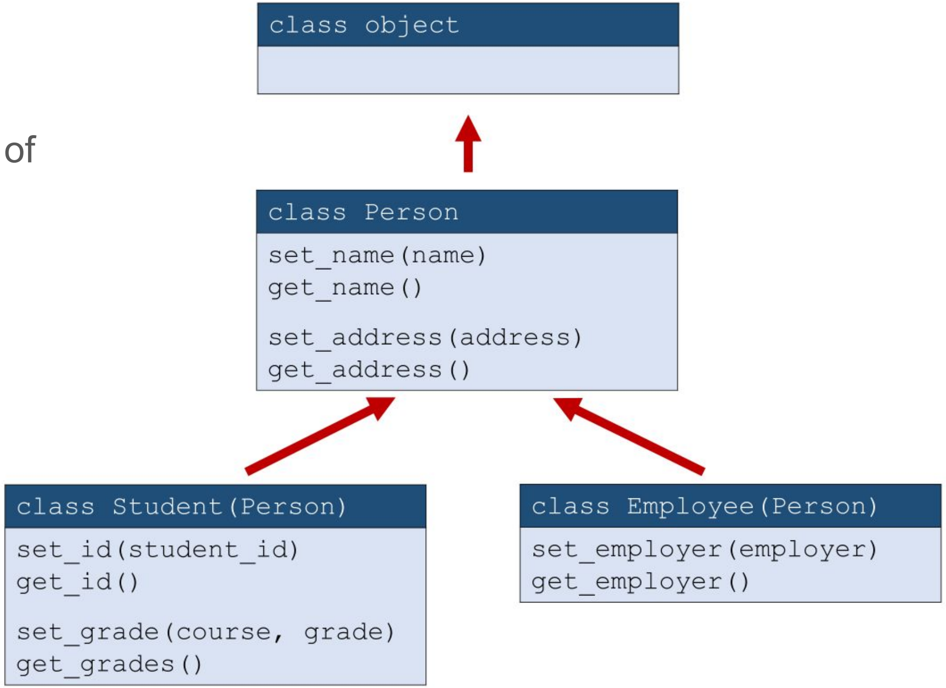


https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Classes in a hierarchy can be composed using inheritance

- Parent class (superclass)
- Child class (subclass)
  - inherits all data Person and behaviors of
  - parent class
  - add more info
  - add more behavior
  - override behavior





https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Classes can override inherited attributes

overloading.py

```python
class A:
    def say(self):
        print('A says hello')

class B(A):  # B is a subclass of A
    def say(self):
        print('B says hello')
        super().say()
```

Python shell

```
> B().say()
| B says hello
| A says hello
```

https://gsbrodal.github.io/ipsa/slides/all-slides.pdf

Classes can override inherited attributes

```
class PoliteList(list):

    def __init__(self):

        print("Thanks for creating me!")

    def __repr__(self):

        return "Polite list = " + list.__repr__(self)

    def __setitem__(self, index, value):

        print(f”I will now set the {index}th value with {value}”)

        list.__setitem__(self, index, value)

    def __getitem__(self, index):

        print(f"You want {index}th value? Here!")

        return list.__getitem__(self, index)
```

```
>>> x = PoliteList()
"Thanks for creating me!"

>>> x[0] = ‘A’

“I will now set the 0th value
with ‘A’”

>>> x[0]

“You want the 0th value? Here!”

‘A’

>>> print(x)

“Polite list = [‘A’]”
```

Summary

- Everything in python is an object
- Classes are instantiated as objects
- Special methods can be used to control how operators work
- Defining custom classes with custom methods/attributes can be powerful
- Object oriented programming abstracts data and operations in a way that enables complex program functions
- Object hierarchy and inheritance allows us to create flexible class definitions with minimal redundancy

Glossary

- `class` -- The definition used to construct objects. Think of it like a blueprint. This is `class Person` in our code.
- `object` -- Each time you use a `class` it creates an object. This the `becky` variable.
- `instance` -- Another name for an object, as in "this is an instance of a Person."
- `instantiate` -- A way to say "create an object" or "create an instance".
- `attribute` -- Any data that is part of the objects as defined by the `class` you used to create it. This is `self.name` or `self.age` in our code.
- `method` -- It's just a function that's been attached to a class. Don't get confused when people claim a method is radically different from a function. Technically just a type of attribute
- `special/magic/dunder methods` -- methods that are usually not called directly but define operations
- `inheritance` -- This is a complicated topic but you can have a `class` that gets additional features from another class. It's similar to how you inherited certain features from your parents.
- `members` -- The members of a class are just the attributes and methods defined in the class.
- `polymorphism` -- A protocol for what happens when classes of different inheritance are used. This is a complex topic, and for you it is likely more trouble than it's worth!