

# OOP & Exceptions

## Principles of Programming Languages

Colorado School of Mines

<https://lambda.mines.edu>

# Object Oriented Programming

# What is OOP?

- Object oriented programming is a type of programming with objects that maintain internal state and pass messages between each other.
- First appeared in the mid-1960s in Simula: a programming language designed for discrete event simulations
- It should not come as a surprise that OOP was first used for simulations, the real world can easily be modeled using objects that pass messages between each other and maintain state
- Smalltalk (1971) also featured OOP and really caused OOP to kick off. Go to CPW's office sometime and chat with him about Smalltalk: he really likes this language!

# OOP: Common Patterns

*Classes:* Classes provide a definition for the data format and the procedures available for member objects.

*Objects:* Objects are instances of a certain class, that follow the rules defined in the class.

Each of these have a variety of kinds of fields:

*Instance Variables:* Variables which are associated with each individual object, these perform the state-maintenance we need for OOP.

*Class Variables:* Variables which are associated with the class itself, all of the objects of that class share this variable and do not have their own copy.

Likewise class/instance methods refer to the methods available from the class versus the instance.

## OOP: Not just tossing functions in a class

*Object-oriented programming is more than just classes and objects; it's a whole programming paradigm based around objects (data structures) that contain data fields and methods. It is essential to understand this; using classes to organize a bunch of unrelated methods together is not object orientation.*

*—Junade Ali, Mastering PHP Design Patterns*

**Encapsulation** prevents external code from being concerned with the inner workings of an object by allowing the objects methods to define how state is manipulated.

- Some languages distinguish between private and public variables to specify which variables should be able to be modified by other objects, and which ones should not be.
- Python does this by convention: private methods and variables should be prefixed with an underscore.

Often times, one object may share variables and methods with another class.

## Example

An Employee might inherit from a Person as:

- Both have variables to store first and last names
- They share a method to generate a full name

But the Employee also has variables that make HR people happy...

# Polymorphism

Polymorphism allows methods to take objects of different types.

- Instances of subclasses can be called on functions that take their parent's type, for example, if `load_elevator` can be called on a `Person`, `Employee` inherits from `Person`, then `load_elevator` should be able to be called on an `Employee` instance as well.
- Statically typed languages often provide **generics**, which allows a method to be called on multiple types, even if there is not inheritance between them. An example is templates in C++.
- Dynamically typed languages provide a method to check if an object is an instance of another to allow the same function to take multiple types. In Python, this is `isinstance`.



## Polymorphism Techniques: isinstance

In Python, we can use `isinstance` to make our functions take objects of different types:

```
# n can be an int or a float  
def frobnicate(n):  
    if isinstance(n, float):  
        return ...  
    elif isinstance(n, int):  
        return ...  
    raise TypeError("I only take ints and floats")
```

# Polymorphism Techniques: Duck Typing

*If it walks like a duck and quacks like a duck, then it must be a duck!*

**Duck typing** is a specific kind of polymorphism where we accept any object which has certain variables or methods, even if the objects do not have a common parent.

In Python, we can do this with `hasattr`:

```
def f(x):  
    if hasattr(x, "walk") and hasattr(x, "quack"):  
        x.walk()  
        x.quack()  
    else:  
        raise TypeError("I only take ducks!")
```

# OOP in Python: An Intro

To define a class in Python, use the class syntax:

```
class Person:  
    def __init__(self, fname, lname):  
        self.fname = fname  
        self.lname = lname
```

- The `__init__` method is the name of the **magic method** that Python calls to construct an instance of the class.
- To construct an instance, call `Person(fname, lname)`.
- All instance methods (such as `__init__`) take a reference to the instance as their first argument. By convention, this is usually named `self`.

# Magic Methods

**Magic methods** are special method names recognized by Python's internals when it needs to perform a certain action. They start and end with a double underscore.

- `__init__` gets called on newly constructed instances of the object.
- `__del__` gets called when an instance is destructed.
- `__eq__` gets called to test if two instances are equal.
- `__call__` gets called when an instance is called as a function.
- `__getitem__` gets called when an instance gets square brackets (such as to get an item from a list)
- `__repr__` called by the built-in `repr()` function for the "official" string representation of an object.
- `__str__` called by the built-in `str()` function for the "informal", more human-readable, string representation of an object.

There's plenty of more, see "Data Model" in the Python Documentation for more information.

## Example of Using Magic Methods

```
class Employee(Person):  
    ...  
    def __str__(self):  
        name = f'{self.fname} {self.lname}'  
        return f'<Employee name="{name}" salary=${self.salary}>'
```

Then, to use this:

```
>>> s = Employee('Linus Torvalds', 250000)  
>>> print(str(s))  
<Employee name="Linus Torvalds" salary=$250000>
```

# Inheritance in Python

The type written in parentheses after the name defines the **base class**. This class will inherit from the base class.

```
class Employee(Person):  
    def __init__(self, ssn, account, *args, **kwargs):  
        self.ssn = ssn  
        self.account = account  
        super().__init__(*args, **kwargs)
```

- Employee inherits from Person
- `super()` refers to the object in its base class. In this case, we call the `init` of our base class with the remaining arguments.

# Multiple Inheritance

Unlike Java, Python classes can inherit from multiple base classes, allowing common variables and methods from two classes. For example (maybe in a GUI toolkit):

```
class Button(Rectangle, Clickable, Labeled):
```

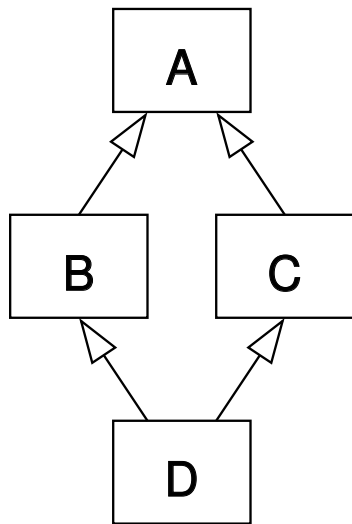
```
    ...
```

# The Diamond Problem

Suppose we have a base class A. We also have subclasses B and C that inherit from A. D inherits from both B and C.

If B and C both override the same method from A, which method does D get?

- Python and Perl's solution: take the method from the first base class listed when the class was declared.
- C++: requires explicit statement of which class the method will dispatch from.
- Java: only allows single inheritance.





# Error Handling

Our programs may encounter errors, for example, if a file was not found, or we were not able to connect to a remote server. Rather than crashing the entire program, software engineers often desire a way to clean up the error and report an error occurred.

# Old School Error Handling

In a C program, you might handle errors like this:

```
/* Return code -1 = error, and errno will be set */
int f(int x) {
    char *A = malloc(x);
    /* ... */
    int pid = fork();
    if (pid == -1)
        goto fail;
    /* ... */
    return z;
fail:
    /* cleanup from error */
    free(A);
    return -1;
}
```

# What's the issue with old school?

- 1 Return codes for errors are never standard, and the programmer always needs to look up what a function call might return.
- 2 If a programmer forgets to handle an error from a function they call, it could have undesired results.
- 3 Functions which handle errors that just want to pass the error up to their caller have to have this error handling code, even though they are not using it directly.

# Go's Error Handling

In Go, functions that return errors return two values: the result, and an error object:

```
f, err := os.Open("filename.ext")
```

It is then the programmer's responsibility to handle the error, potentially returning the error to the caller:

```
if err != nil {  
    return err  
}
```

# Is Go Any Better?

- 1 Return codes for errors are never standard, and the programmer always needs to look up what a function call might return. **Taken care of using error objects!**
- 2 If a programmer forgets to handle an error from a function they call, it could have undesired results. **Still an issue!**
- 3 Functions which handle errors that just want to pass the error up to their caller have to have this error handling code, even though they are not using it directly. **Still an issue!**

# How About Python?

Python provides an **exception handling** system.

- An exception is a special kind of object, like an error object in Go.
- Exceptions are *raised*, not returned.
- A try/except block can be used to intercept an exception, and preform cleanup from the error.
- If function only cares about passing the error to their caller, they need not write any error handling code just to pass it to their caller.

## Try/Except Example

```
def graph(data):
    for point in data:
        if not isinstance(point, GraphPoint):
            raise TypeError('need GraphPoint data')
    ...

def graph_string(s):
    data = user_convert(s)
    return graph(data)

def graph_from_user():
    while True:
        try:
            graph_string(input('point>'))
        except TypeError as e:
            print(e)
```



# Try/Except/Else

An else block placed after a try/except will be executed only if the exception did not occur:

```
while True:
    try:
        f = open(name)
    except FileNotFoundError as e:
        os.chdir('..')
    else:
        print('File Opened!')
        break
```

# Try/Except/Finally

Code listed in the finally block of a try/except/(else) will always be executed, regardless of whether an exception occurs.

```
# note that this "GUIDialog" is made up, but this  
# presents a real-world type usage  
def promptyn(prompt):  
    try:  
        g = GUIDialog(buttons=YESNO, text=prompt)  
        g.show()  
        return g.response()  
    except ValueError:  
        logger.log("Unable to set prompt string")  
    finally:  
        g.close()
```

Note that the context manager (with statement) in Python can often be used as a substitute for a finally.

# Custom Exceptions

Python has a large set of built-in exceptions, but when one of the built-ins is insufficient, you can subclass `Exception` and make your own.

```
class PlottingError(Exception):  
    pass
```

Note that you should usually try and use one of exceptions the language provides rather than define your own.

# Is Python Any Better?

- 1 Return codes for errors are never standard, and the programmer always needs to look up what a function call might return. **Solved!**
- 2 If a programmer forgets to handle an error from a function they call, it could have undesired results. **Solved!**
- 3 Functions which handle errors that just want to pass the error up to their caller have to have this error handling code, even though they are not using it directly. **Solved!**