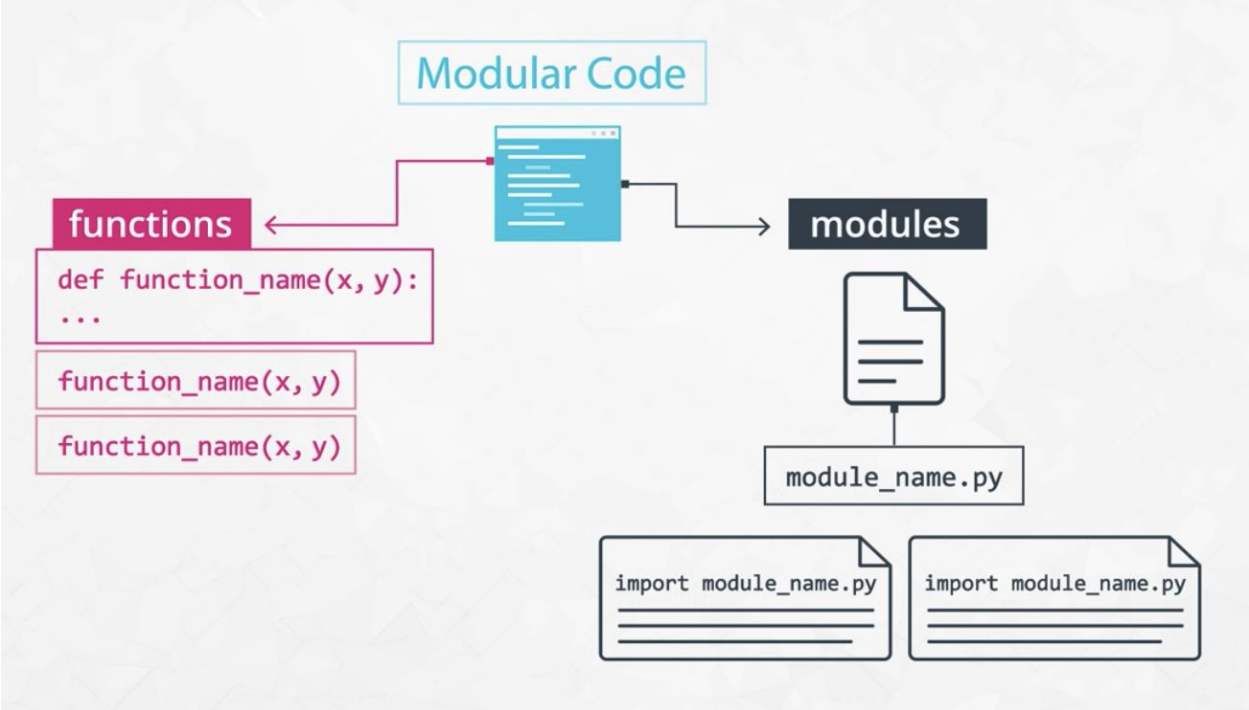
Software Engineering Fundamentals

Clean and Modular Code

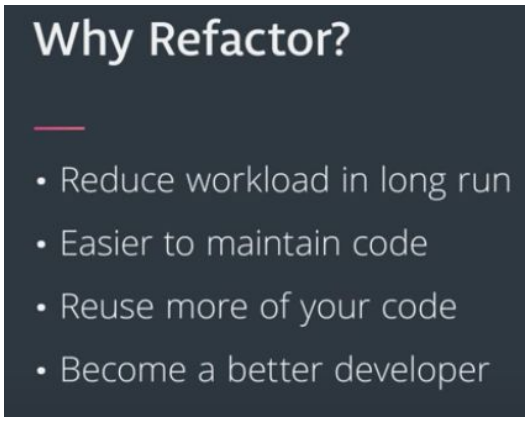


- **PRODUCTION CODE:** software running on production servers to handle live users and data of the intended audience. Note this is different from production quality code, which describes code that meets expectations in reliability, efficiency, etc., for production. Ideally, all code in production meets these expectations, but this is not always the case.
- **CLEAN:** readable, simple, and concise. A characteristic of production quality code that is crucial for collaboration and maintainability in software development.
- **MODULAR:** logically broken up into functions and modules. Also an important characteristic of production quality code that makes your code more organized, efficient, and reusable.
- **MODULE:** a file. Modules allow code to be reused by encapsulating them into files that can be imported into other files.

Refactoring Code

- **REFACTORING:** restructuring your code to improve its internal structure, without changing its external functionality. This gives you a chance to clean and modularize your program after you've got it working.
- Since it isn't easy to write your best code while you're still trying to just get it working, allocating time to do this is essential to producing high quality code. Despite the initial

time and effort required, this really pays off by speeding up your development time in the long run.
- You become a much stronger programmer when you're constantly looking to improve your code. The more you refactor, the easier it will be to structure and write good code the first time.



Writing Clean Code: Meaningful Names

- Be descriptive and imply type - E.g. for booleans, you can prefix with is_ or has_ to make it clear it is a condition. You can also use part of speech to imply types, like verbs for functions and nouns for variables.
- Be consistent but clearly differentiate - E.g. age_list and age is easier to differentiate than ages and age.
- Avoid abbreviations and especially single letters - (Exception: counters and common math variables) Choosing when these exceptions can be made can be determined based on the audience for your code. If you work with other data scientists, certain variables may be common knowledge. While if you work with full stack engineers, it might be necessary to provide more descriptive names in these cases as well.
- Long names != descriptive names - You should be descriptive, but only with relevant information. E.g. good functions names describe what they do well without including details about implementation or highly specific uses.

Writing Clean Code: Nice Whitespace

- Organize your code with consistent indentation - the standard is to use 4 spaces for each indent. You can make this a default in your text editor.
- Separate sections with blank lines to keep your code well organized and readable.
- Try to limit your lines to around 79 characters, which is the guideline given in the PEP 8 style guide. In many good text editors, there is a setting to display a subtle line that indicates where the 79 character limit is.

Writing Modular Code

- **Tip: DRY (Don't Repeat Yourself)**
    - Don't repeat yourself! Modularization allows you to reuse parts of your code. Generalize and consolidate repeated code in functions or loops.
- **Tip: Abstract out logic to improve readability**
    - Abstracting out code into a function not only makes it less repetitive, but also improves readability with descriptive function names. Although your code can become more readable when you abstract out logic into functions, it is possible to over-engineer this and have way too many modules, so use your judgement.
- **Tip: Minimize the number of entities (functions, classes, modules, etc.)**
    - There are tradeoffs to having function calls instead of inline logic. If you have broken up your code into an unnecessary amount of functions and modules, you'll have to jump around everywhere if you want to view the implementation details for something that may be too small to be worth it. Creating more modules doesn't necessarily result in effective modularization.
- **Tip: Functions should do one thing**
    - Each function you write should be focused on doing one thing. If a function is doing multiple things, it becomes more difficult to generalize and reuse. Generally, if there's an "and" in your function name, consider refactoring.
- **Tip: Arbitrary variable names can be more effective in certain functions**
    - Arbitrary variable names in general functions can actually make the code more readable.
- **Tip: Try to use fewer than three arguments per function**
    - Try to use no more than three arguments when possible. This is not a hard rule and there are times it is more appropriate to use many parameters. But in many cases, it's more effective to use fewer arguments. Remember we are modularizing to simplify our code and make it more efficient to work with. If your function has a lot of parameters, you may want to rethink how you are splitting this up.

Efficient Code

Knowing how to write code that runs efficiently is another essential skill in software development. Optimizing code to be more efficient can mean making it:

- Execute faster
- Take up less space in memory/storage

The project you're working on would determine which of these is more important to optimize for your company or product. When we are performing lots of different transformations on large amounts of data, this can make orders of magnitudes of difference in performance.

Testing

Testing your code is essential before deployment. It helps you catch errors and faulty conclusions before they make any major impact. Today, employers are looking for data scientists with the skills to properly prepare their code for an industry setting, which includes testing their code.

Testing And Data Science

- Problems that could occur in data science aren't always easily detectable; you might have values being encoded incorrectly, features being used inappropriately, unexpected data breaking assumptions
- To catch these errors, you have to check for the quality and accuracy of your analysis in addition to the quality of your code. Proper testing is necessary to avoid unexpected surprises and have confidence in your results.
- **TEST DRIVEN DEVELOPMENT:** a development process where you write tests for tasks before you even write the code to implement those tasks.
- **UNIT TEST:** a type of test that covers a "unit" of code, usually a single function, independently from the rest of the program.

Unit and Integration Tests

We want to test our functions in a way that is repeatable and automated. Ideally, we'd run a test program that runs all our unit tests and cleanly lets us know which ones failed and which ones succeeded. Fortunately, there are great tools available in Python that we can use to create effective unit tests!

What is the Unit Test?

Unit Tests are conducted by developers and test the unit of code(aka module, component) he or she developed. It is a testing method by which individual units of source code are tested to determine if they are ready to use. It helps to reduce the cost of bug fixes since the bugs are identified during the early phases of the development lifecycle.

Unit Test Advantages and Disadvantages

The advantage of unit tests is that they are isolated from the rest of your program, and thus, no dependencies are involved. They don't require access to databases, APIs, or other external sources of information. However, passing unit tests isn’t always enough to prove that our program is working successfully. To show that all the parts of our program work with each other properly, communicating and transferring data between them correctly, we use integration tests.

What is an Integration Test?

Integration testing is executed by testers and tests integration between software modules. It is a software testing technique where individual units of a program are combined and tested as a group. Test stubs and test drivers are used to assist in Integration Testing. Integration tests are performed in two ways, they are a bottom-up method and the top-down method.

KEY DIFFERENCES

- Unit testing is a testing method by which individual units of source code are tested to determine if they are ready to use, whereas Integration testing checks integration between software modules.

- Unit Testing tests each part of the program and shows that the individual parts are correct, whereas Integration Testing combines different modules in the application and tests as a group to see they are working fine.
- Unit Testing starts with the module specification, while Integration Testing starts with interface specification.
- Unit Testing can be performed at any time, on the other hand, Integration Testing is performed after unit testing and before system testing.
- Unit Testing is executed by the developer, whereas Integration Testing is performed by the testing team.
- Unit Testing errors can be found easily, whereas Integration Testing it is difficult to find errors.
- Unit Testing is a kind of white box testing, whereas Integration Testing is a kind of black-box testing.

Pytest

pytest is a framework that makes building simple and scalable tests easy. Tests are expressive and readable—no boilerplate code required. Get started in minutes with a small unit test or complex functional test for your application or library.

Test Driven Development and Data Science

- **TEST DRIVEN DEVELOPMENT:** writing tests before you write the code that's being tested. Your test would fail at first, and you'll know you've finished implementing a task when this test passes.
- Tests can check for all the different scenarios and edge cases you can think of, before even starting to write your function. This way, when you do start implementing your function, you can run this test to get immediate feedback on whether it works or not in all the ways you can think of, as you tweak your function.
- When refactoring or adding to your code, tests help you rest assured that the rest of your code didn't break while you were making those changes. Tests also help ensure that your function behavior is repeatable, regardless of external parameters, such as hardware and time.

Log Messages

Logging is the process of recording messages to describe events that have occurred while running your software.

Logging is the process of recording messages to describe events that have occurred while running your software. Let's take a look at a few examples, and learn tips for writing good log messages.

- **Tip: Be professional and clear**

  Bad: Hmmm... this isn't working???

  Bad: idk.... :(

  Good: Couldn't parse file.

- **Tip: Be concise and use normal capitalization**

  Bad: Start Product Recommendation Process

  Bad: We have completed the steps necessary and will now proceed with the recommendation process for the records in our product database.

  Good: Generating product recommendations.

- **Tip: Choose the appropriate level for logging**

  DEBUG - level you would use for anything that happens in the program.

  ERROR - level to record any error that occurs

  INFO - level to record all actions that are user-driven or system specific, such as regularly scheduled operations

- **Tip: Provide any useful information**

  Bad: Failed to read location data

  Good: Failed to read location data: store_id 8324971

Code Reviews

Code reviews benefit everyone in a team to promote best programming practices and prepare code for production.

Submitting pull requests

Review your own code

You should read over your own code, make sure you didn't omit any files, check for style errors and so on. You should do this before you create the PR.

Keep your changes small

Small changes are much easier to review. They're also much easier to explain to people. This means that mistakes are less likely to sneak through and the discussion of the change should be much more productive. It also means your PR should avoid ending up with tens of comments that go round and round in circles without ever getting your code merged. It is not always possible to break a change up into small parts but it very often is. A good guideline for this is that your change should not affect a combined total of more than 400 lines.

Write good descriptions

The title of the PR should include the reference for any tickets covered by this change. You may also want to copy the description from the ticket into your description. The description for your PR should tell the story of your change in a clear and concise way. It should explain both why you are making this change and what you changed. Not all reviewers will have a deep knowledge of that part of the system so you must make it possible for them to provide useful insight. This means explaining what the existing behaviour was, how you changed it, and why this was thought to be useful.

Write good commit messages

Your commits should be logically separate changes. This means that you should aim to separate things like lint cleanups from actual behaviour changes. They should have good commit messages that explain what was changed at each step.

Respond quickly

You shouldn't go off and get stuck into something else while you have PRs open. This makes the reviews less effective for everyone because everyone will now need to switch context more frequently. You will need to remember what was going on in your change, they will need to familiarise themselves with what was going on when they raised a comment. If you have a PR open your number 1 priority should be getting it reviewed and merged.

Pair review

If you're working on a component or a service that others are not familiar with then offer to do an in-person pair review with someone. This guarantees you'll get eyes on your changes and get your code through the door quicker. It's also a great way for knowledge sharing and teaching people new techniques.

Reviewing code

Know what you need to review

Because getting PRs merged quickly is so important you need to stay on top of the changes people want your input on. You can use tools like Trailer.app or even simply searches in GitHub using "involves:<YOUR USERNAME>" to find PRs relevant to you.

Respond quickly

Tolerate being interrupt driven. You need focussed time to your other work done but PRs are time sensitive because they block other people. You shouldn't put off code review for more than a few hours. Never more than a day.

Prioritise code review highly

It's one of the most important things you can work on. The only things that should come ahead of it are time critical work such as fixing availability problems or dealing with high priority requests from customers. This means you should expect to spend some time every day doing reviews and that you will probably need to spend 2-3 sessions per day replying to PRs and reading other people's code.

Be thorough

Reviewing code is hard and error prone. It is our last line of defence against downtime and tech debt. You must pay attention, ask questions and not +1 lightly. Sometimes you will slip and miss something or only notice late in the review process. This not great but it is forgiven. Own up to it and move on. Confused about a bit of code? Ask what it does - there are no stupid questions.

Don't block progress

While code reviews need to be done thoughtfully and thoroughly we also need to avoid blocking progress. This means you should help people with solutions, not only identify problems. Often it might be faster to go and pair with someone for a bit to get their code tidied up instead of going back and forth on GitHub about it. Be aware that some test suites takes some time to run and it may make more sense for some fixes to be made in subsequent PRs instead of being added to this one.

Things to look for

Clarity

Things should be named well and should be easy to follow when reading. The code should attempt to be self documenting.

Correctness

There must be unit tests. They should test the edge cases. The code should behave as the submitter described. The code should use other APIs correctly.

Security

The design should not introduce any security problems such as potential denial of service attacks or unintended information disclosures. In particular we should be aware of potential CSRF and XSS attacks.

Performance

The code should perform within our targets for a particular area. It should not use obviously suboptimal algorithms. However optimisation is usually best left to later. Except when it can also improve other areas at the same time. Simpler code is often faster.

Questions to Ask Yourself When Conducting a Code Review

First, let's look over some of the questions we may ask ourselves while reviewing code. These are simply from the concepts we've covered in these last two lessons!

Is the code clean and modular?

- Can I understand the code easily?
- Does it use meaningful names and whitespace?

- Is there duplicated code?
- Can you provide another layer of abstraction?
- Is each function and module necessary?
- Is each function or module too long?

Is the code efficient?

- Are there loops or other steps we can vectorize?
- Can we use better data structures to optimize any steps?
- Can we shorten the number of calculations needed for any steps?
- Can we use generators or multiprocessing to optimize any steps?

Is documentation effective?

- Are in-line comments concise and meaningful?
- Is there complex code that's missing documentation?
- Do functions use effective docstrings?
- Is the necessary project documentation provided?

Is the code well tested?

- Does the code high test coverage?
- Do tests check for interesting cases?
- Are the tests readable?
- Can the tests be made more efficient?

Is the logging effective?

- Are log messages clear, concise, and professional?
- Do they include all relevant and useful information?
- Do they use the appropriate logging level?

Why Object-Oriented Programming?

Object-oriented programming has a few benefits over procedural programming, which is the programming style you most likely first learned. As you'll see in this lesson,

- object-oriented programming allows you to create large, modular programs that can easily expand over time;
- object-oriented programs hide the implementation from the end-user.

Consider Python packages like Scikit-learn, pandas, and NumPy. These are all Python packages built with object-oriented programming. Scikit-learn, for example, is a relatively large and complex package built with object-oriented programming. This package has expanded over the years with new functionality and new algorithms.

When you train a machine learning algorithm with Scikit-learn, you don't have to know anything about how the algorithms work or how they were coded. You can focus directly on the modeling.

Here's an example taken from the Scikit-learn website:

```
from sklearn import svm
X = [[0, 0], [1, 1]]
y = [0, 1]
clf = svm.SVC()
clf.fit(X, y)
```

How does Scikit-learn train the SVM model? You don't need to know because the implementation is hidden with object-oriented programming. If the implementation changes, you as a user of Scikit-learn might not ever find out. Whether or not you SHOULD understand how SVM works is a different question.

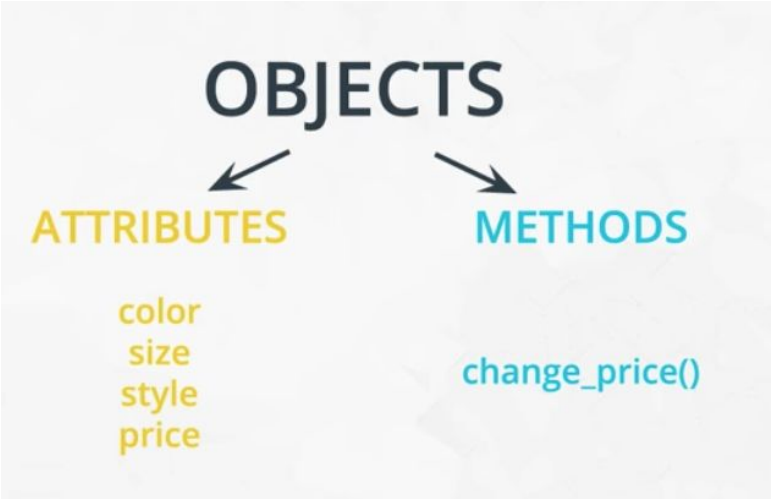
Procedural vs Object-Oriented Programming

Procedural programming is essentially a list of instructions that get executed one then the other starting from top to bottom of the file.

Object Oriented Programming(OOP) is built around the objects.

OOP will focus on individual characteristics of each object and what they can do. Object has two essential parts; characteristics and actions.

Class, object, method, attribute



In this example yellow and green shirts have the same attributes and methods. They came from a blueprint. A generic shirt has all the attributes and methods. A class is a blueprint consisting of attributes and methods.

By using the same class we can create different objects as we want.

Object-Oriented Programming (OOP) Vocabulary

- class - a blueprint consisting of methods and attributes
- object - an instance of a class. It can help to think of objects as something in the real world like a yellow pencil, a small dog, a blue shirt, etc. However, as you'll see later in the lesson, objects can be more abstract.
- attribute - a descriptor or characteristic. Examples would be color, length, size, etc. These attributes can take on specific values like blue, 3 inches, large, etc.
- method - an action that a class or object could take
- OOP - a commonly used abbreviation for object-oriented programming
- encapsulation - one of the fundamental ideas behind object-oriented programming is called encapsulation: you can combine functions and data all into a single entity. In object-oriented programming, this single entity is called a class. Encapsulation allows you to hide implementation details much like how the scikit-learn package hides the implementation of machine learning algorithms.

Object-Oriented Programming Syntax

Here is a reminder of how a class, object, attributes and methods relate to each other.

```
class Shirt:
    def __init__(self, shirt_color, shirt_size, shirt_style, shirt_price):
        self.color = shirt_color
        self.size = shirt_size
        self.style = shirt_style
        self.price = shirt_price

    def change_price(self, new_price):
        self.price = new_price

    def discount(self, discount):
        return self.price * (1 - discount)
```

‘def __init__’ initializes the attributes of the class with appropriate values.
‘self’ stores the attributes and makes them available throughout the class. ‘self’ is essentially a dictionary that holds all the attributes and attribute values.

Function vs Method

Why is init not a function?
A function and a method look very similar. They both use the def keyword. They also have inputs and return outputs. The difference is that a method is inside of a class whereas a function is outside of a class.

When we instantiate an object, it returns the location of the object. In order to access its attributes we need to assign it to a variable. After assigning into the variable, we can access attributes with dictionary key-value pairs.

```
Shirt('red', 'S', 'short sleeve', 15)
```

```
<__main__.Shirt at 0x7f3ec05a7048>
```

```
new_shirt = Shirt('red', 'S', 'short sleeve', 15)
```

```
print(new_shirt.color)
print(new_shirt.size)
print(new_shirt.style)
print(new_shirt.price)
```

```
red
S
short sleeve
15
```

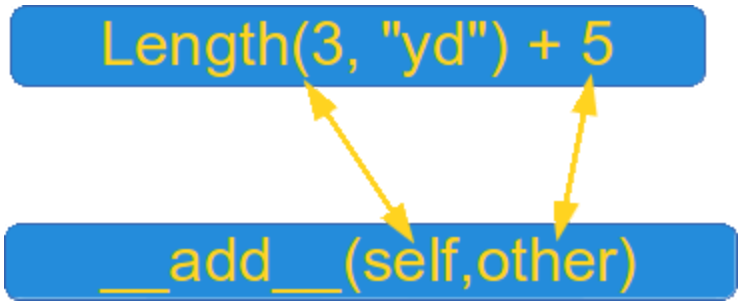
Set and Get methods

The general object-oriented programming convention is to use methods to access attributes or change attribute values. These methods are called set and get methods or setter and getter methods. A get method is for obtaining an attribute value. A set method is for changing an attribute value.

Magic Methods

Magic methods override default Python behaviour.

It's even possible to overload the "+" operator as well as all the other operators for the purposes of your own class. To do this, you need to understand the underlying mechanism. There is a special (or a "magic") method for every operator sign. The magic method for the "+" sign is the __add__ method.

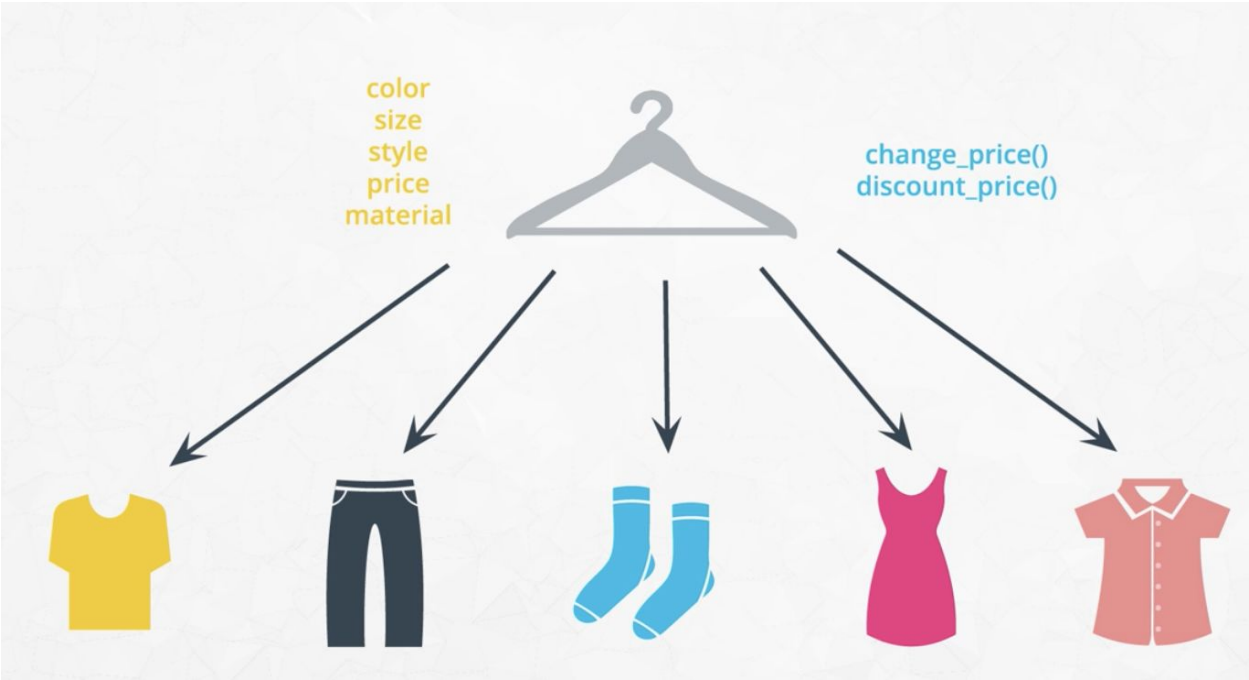


Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.



Clothing is the parent Class to be inherited from. Shirt and Pants are the child classes which inherit. When we create a child class that will inherit from other classes, we need to define the parent class in parentheses.

Shirt initializes itself by using the Clothing method. After the initialization we can add additional attributes and methods.

Also we can override parent classes methods in child methods.

When we add additional methods and attributes to the parent class, they will be automatically inherited in the child class.

```
class Clothing:

    def __init__(self, color, size, style, price):
        self.color = color
        self.size = size
        self.style = style
        self.price = price

    def change_price(self, price):
        self.price = price

    def calculate_discount(self, discount):
        return self.price * (1 - discount)
```

```
class Shirt(Clothing):

    def __init__(self, color, size, style, price, long_or_short):

        Clothing.__init__(self, color, size, style, price)
        self.long_or_short = long_or_short

    def double_price(self):
        self.price = 2*self.price

class Pants(Clothing):

    def __init__(self, color, size, style, price, waist):

        Clothing.__init__(self, color, size, style, price)
        self.waist = waist

    def calculate_discount(self, discount):
        return self.price * (1 - discount / 2)
```

Advanced OOP Topics

Inheritance is the last object-oriented programming topic in the lesson. Thus far you've been exposed to:

- classes and objects
- attributes and methods
- magic methods

- inheritance

Classes, objects, attributes, methods, and inheritance are common to all object-oriented programming languages.

Knowing these topics is enough to start writing object-oriented software. What you've learned so far is all you need to know to complete this OOP lesson. However, these are only the fundamentals of object-oriented programming.

Here is a list of resources for advanced Python object-oriented programming topics.

- class methods, instance methods, and static methods - these are different types of methods that can be accessed at the class or object level
- class attributes vs instance attributes - you can also define attributes at the class level or at the instance level
- multiple inheritance, mixins - A class can inherit from multiple parent classes
- Python decorators - Decorators are a short-hand way for using functions inside other functions

What is pip?

Pip is a Python package manager that helps with installing and uninstalling Python packages. When you execute a command like pip install numpy, pip will download the package from a Python package repository called PyPi.

Virtual Environments

A virtual environment is a silo-ed Python installation apart from your main Python installation. That way you can install packages and delete the virtual environment without affecting your main Python installation.

Instructions for venv

Here are instructions about how to set up virtual environments on a macOS, Linux, or Windows machine using the terminal: instructions link.

These are a few notes for understanding the tutorial:

- If you are using Python 2.7.9 or later (including Python 3), the Python installation should already come with the Python package manager called pip. There is no need to install it.

- env is the name of the environment you want to create. You can call env anything you want.
- Python 3 comes with a virtual environment package pre-installed. So instead of typing python3 -m virtualenv env, you can type python3 -m venv env to create a virtual environment.

Once you've activated a virtual environment, you can then use terminal commands to go into the directory where your Python library is stored. And then you can run pip install .. You'll see that creating a virtual environment actually creates a new folder containing a Python installation. Deleting this folder will remove the virtual environment.