Data Science Nanodegree:

Module 2:

## Lesson 1: The Data Science process:

CRISP-M: Cross industry standard process for Data Mining:

* Business Understanding
* Data Understanding
* Data preparation
* Modeling
* Evaluation
* Deployment

1. **Business Understanding** - this means understanding the problem and questions you are interested in tackling in the context of whatever domain you're working in. Examples include:

* How do we acquire new customers?
* Does a new treatment perform better than an existing treatment?
* How can improve communication?
* How can we improve travel?
* How can we better retain information?

#### Business Questions (Stack Overflow survey)

1. How do I break into the field?
2. What are the placement and salaries of those who attended a coding bootcamp?
3. How well can we predict an individual's salary? What aspects correlate well to salary?
4. How well can we predict an individual's job satisfaction? What aspects correlate well to job satisfaction?



1. **Data Understanding** - at this step, you need to move the questions from **Business Understanding** to data. You might already have data that could be used to answer the questions, or you might have to collect data to get at your questions of interest. Some businesses try to collect data as much as possible then look insight of them to understand more.
2. **Data Preparation** - Luckily stack-overflow has already collected the data for us. However, we still need to wrangle the data in a way for us to answer our questions. The wrangling and cleaning process is said to take 80% of the time of the data analysis process. You will see that will hold true through this lesson, as a majority of the remaining parts of this lesson will be around basic data wrangling strategies.

#### CRISP-DM

In working with missing values, categorical variables, and building out your model, it was probably easy to lose sight of the big picture of the process. Let's take a quick second to recap that here, and pull together the results you should have arrived through your analysis.

1. Business Understanding

1. How do I break into the field?
2. What are the placement and salaries of those who attended a coding bootcamp?
3. How well can we predict an individual's salary? What aspects correlate well to salary?
4. How well can we predict an individual's job satisfaction? What aspects correlate well to job satisfaction?

2. Data Understanding

Here we used the stackoverflow data to attempt to answer our questions of interest. We did 1. and 2. in tandem in this case, using the data to help us arrive at our questions of interest. This is one of two methods that is common in practice. The second method that is common is to have certain questions you are interested in answering, and then having to collect data related to those questions.

3. Prepare Data

This is commonly denoted as 80% of the process. You saw this especially when attempting to build a model to predict salary, and there was still much more you could have done. From working with missing data to finding a way to work with categorical variables, and we didn't even look for outliers or attempt to find points we were especially poor at predicting. There was ton more we could have done to wrangle the data, but you have to start somewhere, and then you can always iterate.

4. Model Data

We were finally able to model the data, but we had some back and forth with step 3. before we were able to build a model that had okay performance. There still may be changes that could be done to improve the model we have in place. From additional feature engineering to choosing a more advanced modeling technique, we did little to test that other approaches were better within this lesson.

5. Results

Results are the findings from our wrangling and modeling. Below are some questions to recap the results found so far.

### Related to Marketing Yourself

1. Catchy image followed by a catchy title.
2. Write in small blocks of text, as large blocks of text will exhaust and demotivate your reader.
3. Start with an engaging question or current event that your audience is likely thinking about.
4. End with a recap and a call to action.

### Related to Keeping Yourself and Your Reader Motivated

1. Keep your post short and sweet. 1-2 pages per post will increase audience engagement, as well as the likelihood you complete the writing of your post.
2. Create an outline. A useful outline for your post could just be: introduction, each of your questions, and then a conclusion. This will help you stay focused when writing your post.
3. Review. Review. Review. When you think you have reviewed enough, find someone else to review. The more review you have of your work, the better it will become.

**The three Text portions of this class contain A LOT of useful information. Use them for reference as you complete your project, and as you continue to write more posts in the future!**

<https://medium.com/@josh_2774/how-do-you-become-a-developer-5ef1c1c68711>

<https://review.udacity.com/#!/rubrics/1507/view>

Congratulations! You've just been put in charge of technical recruiting at Globex, a multinational high- tech firm. This job comes with a corner office, and you have an experienced staff of recruiters at your disposal. They want to know what they should prioritize when recruiting software developers. How important should each of the following be in Globex's hiring process? Knowledge of algorithms and data structures

Congratulations! You've just been put in charge of technical recruiting at Globex, a multinational high- tech firm. This job comes with a corner office, and you have an experienced staff of recruiters at your disposal. They want to know what they should prioritize when recruiting software developers. How important should each of the following be in Globex's hiring process? Experience with specific tools (libraries, frameworks, etc.) used by the employer

Sample of passed projects:

<https://github.com/lakshman533/Brain-Stroke>

Job satisfaction for developers

# Module 3: Software Engineer

## How this Course is Organized

* Software Engineering Practices Part 1 covers how to write well documented, modularized code.
* Software Engineering Practices Part 2 discusses testing your code and logging.
* Introduction to Object-Oriented Programming gives you an overview of this programming style and prepares you to write your own Python package.
* Introduction to Web Development covers building a web application data dashboard.

## Lesson 2: Part 1, how to write well documented, modularized code.

* *Production code*: Software running on production servers to handle live users and data of the intended audience. Note that this is different from *production-quality code*, which describes code that meets expectations for production in reliability, efficiency, and other aspects. Ideally, all code in production meets these expectations, but this is not always the case.
* *Clean code*: Code that is readable, simple, and concise. Clean production-quality code is crucial for collaboration and maintainability in software development.
* *Modular* code: Code that is logically broken up into functions and modules. Modular 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 have got it working.
* Since it is not easy to write your best code while you are 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 are 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

Use meaningful names.

* Be descriptive and imply type: For booleans, you can prefix with is\_ or has\_ to make it clear it is a condition. You can also use parts of speech to imply types, like using verbs for functions and nouns for variables.
* Be consistent but clearly differentiate: age\_list and age is easier to differentiate than ages and age.
* Avoid abbreviations and single letters: You can determine when to make these exceptions 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. (Exceptions include counters and common math variables.)
* Long names aren't the same as descriptive names: You should be descriptive, but only with relevant information. For example, good function names describe what they do well without including details about implementation or highly specific uses.

Try testing how effective your names are by asking a fellow programmer to guess the purpose of a function or variable based on its name, without looking at your code. Coming up with meaningful names often requires effort to get right.

### Writing clean code: Nice whitespace

Use whitespace properly.

* Organize your code with consistent indentation: the standard is to use four 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.

For more guidelines, check out the code layout section of PEP 8 in the following notes.

PEP8 guide:

<https://www.python.org/dev/peps/pep-0008/?#code-lay-out>

### Writing Modular Code

Follow the tips below to write 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 trade-offs 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 when 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. If your function has a lot of parameters, you may want to rethink how you are splitting this up.

### Docstrings

Docstring, or documentation strings, are valuable pieces of documentation that explain the functionality of any function or module in your code. Ideally, each of your functions should always have a docstring.

Docstrings are surrounded by triple quotes. The first line of the docstring is a brief explanation of the function's purpose.

#### One-line docstring

**def** **population\_density**(population, land\_area):

"""Calculate the population density of an area."""

**return** population / land\_area

If you think that the function is complicated enough to warrant a longer description, you can add a more thorough paragraph after the one-line summary.

#### Multi-line docstring

**def** **population\_density**(population, land\_area):

"""Calculate the population density of an area.

Args:

population: int. The population of the area

land\_area: int or float. This function is unit-agnostic, if you pass in values in terms of square km or square miles the function will return a density in those units.

Returns:

population\_density: population/land\_area. The population density of a

particular area.

"""

**return** population / land\_area

The next element of a docstring is an explanation of the function's arguments. Here, you list the arguments, state their purpose, and state what types the arguments should be. Finally, it is common to provide some description of the output of the function. Every piece of the docstring is optional; however, doc strings are a part of good coding practice.

# Welcome To Software Engineering Practices, Part 2

In part 2 of software engineering practices, you'll learn about the following practices of software engineering and how they apply in data science.

* Testing
* Logging
* Code reviews

## 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, or 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 (TDD): A development process in which 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.

### Resources

* Four Ways Data Science Goes Wrong and How Test-Driven Data Analysis Can Help: [**Blog Post**](https://www.predictiveanalyticsworld.com/patimes/four-ways-data-science-goes-wrong-and-how-test-driven-data-analysis-can-help/6947/)
* Ned Batchelder: Getting Started Testing: [**Slide Deck**](https://speakerdeck.com/pycon2014/getting-started-testing-by-ned-batchelder) and [**Presentation Video**](https://www.youtube.com/watch?v=FxSsnHeWQBY)

# Unit 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!

## 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. In this lesson, we'll focus on unit tests; however, when you start building larger programs, you will want to use integration tests as well.

To learn more about integration testing and how integration tests relate to unit tests, see [**Integration Testing**](https://www.fullstackpython.com/integration-testing.html). That article contains other very useful links as well.

### Unit Testing Tools

To install pytest, run pip install -U pytest in your terminal. You can see more information on getting started [**here**](https://docs.pytest.org/en/latest/getting-started.html).

#### Create a test file starting with test\_.

* Define unit test functions that start with test\_ inside the test file.
* Enter pytest into your terminal in the directory of your test file and it detects these tests for you.

test\_ is the default; if you wish to change this, you can learn how in this **[pytest configuration](https://docs.pytest.org/en/latest/customize.html" \t "_blank)**.

In the test output, periods represent successful unit tests and Fs represent failed unit tests. Since all you see is which test functions failed, it's wise to have only one assert statement per test. Otherwise, you won't know exactly how many tests failed or which tests failed.

Your test won't be stopped by failed assert statements, but it will stop if you have syntax errors.

## Test-driven development and data science

* Test-driven development: Writing tests before you write the code that’s being tested. Your test fails at first, and you know you’ve finished implementing a task when the test passes.
* Tests can check for different scenarios and edge cases before you even start to write your function. When start implementing your function, you can run the test to get immediate feedback on whether it works or not 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 helps ensure that your function behavior is repeatable, regardless of external parameters such as hardware and time.

Test-driven development for data science is relatively new and is experiencing a lot of experimentation and breakthroughs. You can learn more about it by exploring the following resources.

* [**Data Science TDD**](https://www.linkedin.com/pulse/data-science-test-driven-development-sam-savage/)
* [**TDD for Data Science**](http://engineering.pivotal.io/post/test-driven-development-for-data-science/)
* [**TDD is Essential for Good Data Science Here's Why**](https://medium.com/@karijdempsey/test-driven-development-is-essential-for-good-data-science-heres-why-db7975a03a44)
* [**Testing Your Code**](http://docs.python-guide.org/en/latest/writing/tests/) (general python TDD)

# Portfolio Exercise: Upload a Package to PyPi

Personal portfolios are an excellent way to demonstrate your knowledge and creativity. In fact, they are little by little becoming a must-have for people working in the tech industry. In this portfolio building exercise, you will build a Python package and upload the package to PyPi.

**NOTE that a portfolio exercise like this is NOT reviewed. So you will not submit your work on this, and you do not need to complete this assignment in order to graduate.**

## Getting Started

Next, you'll find a blank classroom workspace where you can do your work. The benefits of using the workspace over your own computer are two fold:

* you shouldn't run into set-up issues
* if you install something on the workspace that causes issues, you can always reset everything

The workspace is the same Ubuntu Linux environment you've been using to do the exercises. You are also welcome to develop your package locally on your own computer if you prefer.

For local development, if you are developing on macOS, you can use the exact same terminal commands. On a windows machine, the commands are slightly different and you'll need to use the command prompt. This link contains a [**comparison of MS-DOS vs Linux commands**](https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Step_by_Step_Guide/ap-doslinux.html).

If at some point you decide to reset your workspace, make sure you have saved your work somewhere else like in a GitHub repository; otherwise, all of your work will be deleted. You can connect to a remote GitHub repo from within the workspace terminal. Here is a reminder of how to do that: [**adding an existing repository to GitHub using the command line**](https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/). Another option is to download files or folders directly from the workspace. You right click on the file/folder name and click "Download".

In the workspace, you'll see that the files and folders have already been set up for you in terms of the structure, but the files themselves are empty.

Every time you type in the workspace, your work is automatically saved. But you should still save and commit your work to GitHub as well.

## Instructions

Build a Python package and upload the package to PyPi. We encourage you to use object-oriented programming and tackle a problem that interests you. The Introduction to Object-Oriented Programming lesson has all of the information needed to create a package and upload the package. As mentioned in the lesson material, you'll need to use a unique name to upload the package to PyPi since each package needs a unique name; hence 'dsnd-probability' will not work since that package name is already taken.

Here are a few ideas for packages:

* create a package that does basic matrix algebra such as addition, subtraction, multiplication, matrix inversion, etc.
* make a calculus package that implements algorithms such as Newton's method
* create a package built on top of Matplotlib or Seaborn that creates well-formatted, well labeled visualizations
* create an object-oriented package to play a game of tic-tac-toe between two human players.
* or make a number guessing game where the computer randomly chooses an integer and then tells a human player if a guess is higher or lower than the number

Feel free to come up with your own ideas. The main goal is to develop and demonstrate your skills in object-oriented programming and uploading packages to PyPi.

### Reminders

* include a README file detailing the files in your package and how to install the package.
* Comment your code - use docstrings and inline comments where appropriate.
* Refactor code when possible - if you find your functions are getting too long, then refactor your code!
* Use object-oriented programming whenever it makes sense to do so.
* You're encouraged to write unit tests! The coding exercises in this lesson contained unit tests, so you can use those tests as a model for your package.
* Use GitHub for version control, and commit your work often.

As a reminder, your package should be placed in a folder with the following folders and files:

* a folder with the name of your package that contains:
  + the Python code that makes up your package
  + a README.md file
  + an \_\_init\_\_.py
  + license.txt
  + setup.cfg
* setup.py file

## Hints and Helpful Links

Because this exercise requires writing and organizing code in a specific way, you might have to rewatch some of the lecture videos especially the "Putting Code on PyPi" concept.

Before you upload your code to PyPi, you should first pip install the package locally to make sure everything works as expected. The "Making a Package" and "Virtual Environments" lesson concepts should be helpful. You should also consider writing unit tests to test the functionality of your package. In the object-oriented programming lesson workspace, there were unit tests inside the 4a\_binomial\_package folder that you can use to help you get started. Those were in a file called test.py.

The object-oriented programming lesson included a complete, working package called dsnd-probability. We encourage you to code a project from scratch; however, if you get stuck, use the dsnd-probability package code as a template. It already contains all of the necessary files you'll need for creating a package. It's also a simple example of object-oriented code. You can use these files, including the setup.py and setup.cfg files, to help structure your own code.

Those files are located in the "Exercise: Upload to PyPi" section inside the "5\_exercise\_upload\_to\_pypi" folder.

Besides the lesson on object-oriented programming, you might find [**this package building summary guide from the Python website**](https://packaging.python.org/guides/distributing-packages-using-setuptools/) helpful.

For a much more detailed explanation of distributing Python packages, check out the documentation on Distutils.

1. [**Introduction**](https://docs.python.org/3/distutils/introduction.html)
2. [**setup.py script**](https://docs.python.org/3/distutils/setupscript.html)
3. [**config file**](https://docs.python.org/3/distutils/configfile.html)
4. [**source distributions**](https://docs.python.org/3/distutils/sourcedist.html)
5. [**built distributions**](https://docs.python.org/3/distutils/builtdist.html)
6. [**uploading to PyPi**](https://docs.python.org/3/distutils/packageindex.html)

## Uploading to PyPi

When you are ready to upload your package, you can first upload to the **[PyPi test repository](https://test.pypi.org/" \t "_blank)**. Once everything is working as expected, you can upload to the public facing **[PyPi repository](https://pypi.org/" \t "_blank)**.

As a reminder, you'll need to create a username for both the test and public facing repositories. You'll also need to pip install the twine package with: pip install twine. You can rewatch the lesson videos to see how to upload your package.

Continue on to the next sections to get some troubleshooting tips, and access the project workspace. When you're finished with your project, show off your work on a personal website, GitHub, and LinkedIn.

# Natural Language Processing Pipelines

In this lesson, you'll be introduced to some of the steps involved in a NLP pipeline:

1. Text Processing
   * Cleaning
   * Normalization
   * Tokenization
   * Stop Word Removal
   * Part of Speech Tagging
   * Named Entity Recognition
   * Stemming and Lemmatization
2. Feature Extraction
   * Bag of Words
   * TF-IDF
   * Word Embeddings
3. Modeling

## How NLP Pipelines Work

The 3 stages of an NLP pipeline are: Text Processing > Feature Extraction > Modeling.

* **Text Processing:** Take raw input text, clean it, normalize it, and convert it into a form that is suitable for feature extraction.
* **Feature Extraction:** Extract and produce feature representations that are appropriate for the type of NLP task you are trying to accomplish and the type of model you are planning to use.
* **Modeling:** Design a statistical or machine learning model, fit its parameters to training data, use an optimization procedure, and then use it to make predictions about unseen data.

This process isn't always linear and may require additional steps

## Stage 1: Text Processing

The first chunk of this lesson will explore the steps involved in **text processing**, the first stage of the NLP pipeline.

### Why Do We Need to Process Text?

* **Extracting plain text:** Textual data can come from a wide variety of sources: the web, PDFs, word documents, speech recognition systems, book scans, etc. Your goal is to extract plain text that is free of any source specific markup or constructs that are not relevant to your task.
* **Reducing complexity:** Some features of our language like capitalization, punctuation, and common words such as *a, of,* and *the,* often help provide structure, but don't add much meaning. Sometimes it's best to remove them if that helps reduce the complexity of the procedures you want to apply later.

### What Text Processing Will You Do in This Lesson?

You'll prepare text data from different sources with the following text processing steps:

1. **Cleaning** to remove irrelevant items, such as HTML tags
2. **Normalizing** by converting to all lowercase and removing punctuation
3. Splitting text into words or **tokens**
4. Removing words that are too common, also known as **stop words**
5. Identifying different **parts of speech** and **named entities**
6. Converting words into their dictionary forms, using **stemming and lemmatization**

After performing these steps, your text will capture the essence of what was being conveyed in a form that is easier to work with.