DS 5100: Programming for Data Science

Spring 2023

Rafael C. Alvarado

5/8/21

Table of contents

Part I

Welcome

Welcome to website for DS 5100 Programming for Data Science, Spring 2023.

> **Note**
>
> This site contains all the content needed to complete the course. All graded coursework is hosted on the Canvas website.

In this course, you will develop skills in Python and R Programming, as well as how to use the command line and GitHub.

The objective of this course is to introduce essential programming concepts, structures, and techniques.

You will gain confidence in not only reading code, but learning what it means to write good quality code.

Additionally, essential and complementary topics are taught, such as testing and debugging, exception handling, and an introduction to visualization.

1 Syllabus

1.1 Welcome

Welcome to DS 5100 Programming for Data Science! In this course, we will develop skills in Python and R Programming, as well as the command line and GitHub. The objective of the course is to introduce essential programming concepts, structures, and techniques from a data science perspective. You will gain confidence in not only reading code, but learning what it means to write good quality code. Additionally, essential and complementary topics are taught, such as testing and debugging, exception handling, and an introduction to visualization.

1.1.1 The Data Science Perspective

Learning to code from the data science perspective means that we will emphasize the role that code plays in working with the **data pipeline** that underlies nearly everything that a data scientist does, from acquiring data from a variety of sources to cleaning and reshaping it for exploration, analysis, and modeling, to visualizing and interpreting it for the world to understand. At each phase of this process, data are transformed through the medium of code, and so we will always being thinking of a programming language as a data processing language. The meaning of this will become clearer as the course progresses.

1.1.2 Code Fluency

This course is designed to teach the programming knowledge and skills necessary to become an effective data scientist. The focus will be on **code fluency** – the ability to both write and read code, as well as to understand the nature of high quality code. Code fluency encompasses a variety of skills, from the ability to write functions and classes to testing and debugging to packaging and visualizing the results of coding.

Code fluency is important because code is the **primary medium** through which we represent and express our most basic and complex ideas in data science. These ideas include everything from the structure of web pages to the process of back propagation in a neural network. Code is the language with which we represent data and the models that process and interpret data, as well as the data products that make use of our data and the analytical results from it.

1.1.3 Python and R

This course is specifically focused on your ability to read and write code in **Python and R**. It is not a course in computer science or in data wrangling or in software development. Each of those elements will obviously play into our work, but our focus is on the fundamental knowledge of programming – the building blocks from which you can build complex (but not complicated) code to solve real world problems.

1.1.4 Practice Makes Perfect

The guiding philosophy of the course is that coding is a **practice** like many other practices – such as the ability to speak a non-native human language, or to play a musical instrument, or to play a sport, or to perform such as in singing, acting, or dancing. These are all complex practices that involve higher forms of cognitive representation but are also **embodied** practices. This means that they have to be practiced, physically and repetitively, in order for you to be successful at them. Programming languages are like that.

Put another way, **programming is like cooking**, carpentry, and other forms of material creation. Again, in each case high level cognition is involved, but so are the hands and eyes, and an appreciation of the subtle qualities of materials and ingredients is essential to successfully using them to create effective work – a sturdy and beautiful building or a satisfying and exquisite meal.

All of these practices, some of which I am sure each of you has had experience in, are based on the ideas of imitation and drilling which develop into generalization and integration and finally into excellence and mastery of design and execution. Therefore, this course will require the student to observe principles and imitate examples (though writing) on the path to generalization and fluency.

1.2 Learning Goals

Upon completion of this course, you are expected to be able to do the following. In all cases, unless specified, both Python and R are included. In truth, you'll probably learn more than this. :-)

Understand the importance of data and programming for data science

- Understand the relationship between between data and data science.
- Understand how data is related to programming.
- Know broadly what kinds of data exist.

Confidently work in an appropriate programming environment

- Basic operations with Git and GitHub to manage and share your code.
- Confidently write code in Jupyter Lab, Visual Studio Code, and RStudio.
- Understand which editor is appropriate to which task.
- Find and use documents, data, and code online.

Identify and use data types and data structures

- Know the elementary data types for each language: booleans, integers, floats, strings, etc.
- Know the elementary data structures for each language:
  - Python: set, list, dictionary, and tuple.
  - R: vectors, list, matrix, factor.
- Know some of the advanced data structures for each language:
  - Python: Numpy arrays and Pandas series and dataframes.
  - R: dataframes and Tidy tibbles.
- Know and perform basic operations for each data type and structure.
- Select and apply an appropriate data structure based on the problem requirements.

Read and Write to and from various data formats

- Read text and data files from disc.
- Import data into a Pandas and R dataframes

Confidently call and write functions and methods

- Understand the structure and use of functions for programming.
- Use built-in and import functions to perform fundamental tasks.
- Correctly pass parameters and retrieve function output(s).
- Use built-in object methods for data types and structures, e.g. string methods and dataframe methods.
- Know what vectorized functions and methods are.

Confidently write a class and call its methods

- Understand role of classes in organizing code.
- Understand how classes group together variables as attributes and functions as methods into encapsulated components.
- Understand how classes can inherit the variables and methods of other classes.

Use packages to augment existing data structures

- In Python, NumPy and Pandas essentials (e.g. simple queries and small ML computation)
- In Python and R, use a program API to utilize existing functions (e.g. assert statements).

- In R, apply the Tidyverse verbs, such as: `select()`, `filter()`, `arrange()`, `mutate()`, and `summarize()`.
- In R, apply the Tidyverse Pipe operator `%>%` to aggregate data.

Write your own modules of classes in Python

- Write classes and organize them into modules to make your more modular.
- Make your modules sharable so that others can install them with Python's setup and install functions.
- Write documentation for your modules so that others can make sense of them.
- Write test scripts to go with your modules.

Write robust code by implementing the basic principles of program testing and debugging in Python

- Catch errors in your with exception handling and print statements.
- Read error messages produced by the interpreter.
- Fix and harden broken code.

1.3 Assessments

1.3.1 Homework Assignments

Homework assignments will given throughout the course, typically one for each module.

You are encouraged to first try to complete the homework by yourself.

If you work with others, be sure you understand all of the work, and that your final submission is your own work.

Typically, homework assignments that involve Jupyter Notebooks will be submitted through GradeScope as PDFs. However, in some cases the assignment will be submitted through Canvas. In either case, your assignment will be listed in the week's module.

1.3.2 Lateness Policy

Please submit HW assignments on time.

If an issue will prompt late submission, email the TA in advance to explain the situation.

If the HW is submitted late and it is not an excused lateness, 10% of the assignment total points will be deducted per day it is late.

1.3.3 In-Class Activities

During each class there will be activities in which you will write code to demonstrate and extend your knowledge. These may be guided activities or peer-programming exercises. Each of these are designed to exemplify the concepts conveyed in reading and lecture.

Although the results of this work are not graded, you will be graded on your effort to complete them. This will count toward your participation grade.

1.3.4 Quizzes

There will be several quizzes throughout the semester that will assess your knowledge of the various topics. Quizzes are based on the topics and code covered in the readings and activities.

All quizzes are mandatory for all students to take.

Although they can be completed in less time, you have one hour to finish and submit your work.

The quizzes should be done closed book: please do not consult any resources including notes, books, the web, devices, or other external media.

⚠ Warning

Making up missed quizzes is not advised — their timing is part of their value. However, if you know in advance that you will miss any of the scheduled quizzes, you must make arrangements in advance with the instructor. At least one week in advance if possible, or as soon as you are able if an unforeseen event occurs preventing you from taking the quiz.

1.3.5 Course Project

The final project will focus on creating and packaging a module in Python. This module will address a data science problem and be sharable on GitHub (in principle) and installable by others. It will have proper documentation and a testing file.

Project deliverables are due on the last day of course. See Collab for submission details.

More information on the project will be available near the half-way point of the course.

1.3.6 Spirit of the Course

Students must attend each class and participate in group work.

For the programming assignments and quizzes, you must submit your own work.

1.3.7 Submission of Assignments

All assignments must be submitted through Canvas or Gradescope by the specified due dates and times. It is crucial to complete all assigned work—failure to do so will likely result in failing the class.

1.4 Grading

1.4.1 Model

Every assessment is equally important in this course.

Quizes	20%
Homework	20%
Project	20%
Participation	20%
Final Exam	20%

1.4.2 Scale

We follow the 3-4-3 model of grading. That is, within each letter range, the + and − each span 3 values ($[0, 1, 2]$ and $[7, 8, 9]$ respectively), whereas the neutral grade spans the middle 4 values ($[3, 4, 5, 6]$).

Note that it is by convention that we treat 0 in the 1s place as standing for the initial value of a grade span, which leads to the anomaly of the `A` range having 11 values, since it includes both 90 and 100. Past experience shows that treating 90 as a `B+` is considered an outrage by students, so we accept the weirdness :-)

Grade	Min Max
A+	97 - 100
A	93 - 96
A-	90 - 92
B+	87 - 89

```
B         83 - 86
B-        80 - 82 &#8592; minimum passing grade
-----------------
C+        77 - 79
C         73 - 76
C-        70 - 72
```

1.5 Texts

1.5.1 Core Texts

Many of our readings will draw from these texts. We will try to stick with some core texts to provide continuity. These will often be supplemented by shorter sources of information drawn from the web.

- Lutz, 2013, **Learning Python**, 5th Edition, O'Reilly Media.
- McKinney, 2017, **Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython**, 2nd Edition. O'Reilly Media.
- Wickham and Grolemund, 2017, **R for Data Science: Import, Tidy, Transform, Visualize, and Model Data**, 1st Edition. O'Reilly Media.

1.5.2 Other Texts

We occasionally draw from the following texts. They are listed here as supplementary resources that you may want to use later on.

For R

Parts of some of these more be included in various modules.

- Cotton, 2013, **Learning R**, O'Reilly Media.
- Rodríguez, 2021, **Introducing R**, Princeton University faculty website. This a concise website that may want to refer to in your Linear Models course.
- Douglas, et al 2022, **An Introduction to R**, self published.
- Peng, 2020, **R Programming for Data Science**, self published.

For Python

Once you get the hang of Python, you will want to embark on becoming a more effect data science software developer. These books can help.

- Matthis 2023, **Python Crash Course**, 3rd Edition. O'Reilly Media.

- Brett Slatkin, 2019, **Effective Python: 90 Specific Ways to Write Better Python, 2nd Edition**, Addison-Wesley.
- Katz, Philipp and David Katz, 2019, **Learn Python by Building Data Science Applications,** Packt Publishing.
- Lee Vaughan, 2020, **Real-World Python**, No Starch Press.

1.5.3 Access to materials

This course uses a number of books from the O'Reilly Media's online library. This is a commercial site, but as students of UVA, you have free access to it. To access the collection, first you must create an account on the site. See the document Setting up a student account on O'Reilly's Site for help.

1.5.4 Websites

- Python's official documentation
- Python's official tutorial
- R's official documentation
- W3Schools Python Tutorial
- W3Schools R Tutorial
- GeeksForGeeks on Python
- GeeksForGeeks on R
- Tutorialspoint on Python
- Tutorialspoint on R

1.5.5 Cheatsheets

- Python Cheatsheets
- RStudio Cheatsheets

1.5.6 Books to Broaden Your Horizons

- Graham, 2010, Hackers & Painters: Big Ideas from the Computer Age
- Brooks, 1995, The Mythical Man Month
- Shetterly, 2016, Hidden Figures: The American Dream and the Untold Story of the Black Women Mathematicians Who Helped Win the Space Race

1.6 Academic Integrity

The School of Data Science relies upon and cherishes its community of trust. We firmly endorse, uphold, and embrace the University's Honor principle that students will not lie, cheat, or steal, nor shall they tolerate those who do. We recognize that even one honor infraction can destroy an exemplary reputation that has taken years to build. Acting in a manner consistent with the principles of honor will benefit every member of the community both while enrolled in the School of Data Science and in the future.

Students are expected to be familiar with the university honor code, including the section on academic fraud.

Each assignment will describe allowed collaborations, and deviations from these will be considered Honor violations. If you have questions on what is allowable, ask! Unless otherwise noted, exams and individual assignments will be considered pledged that you have neither given nor received help. (Among other things, this means that you are not allowed to describe problems on an exam to a student who has not taken it yet. You are not allowed to show exam papers to another student or view another student's exam papers while working on an exam.) Sending, receiving or otherwise copying electronic files that are part of course assignments are not allowed collaborations (except for those explicitly allowed in assignment instructions).

Assignments or exams where honor infractions or prohibited collaborations occur will receive a zero grade for that entire assignment or exam. Such infractions will also be submitted to the Honor Committee if that is appropriate. Students who have had prohibited collaborations may not be allowed to work with partners on remaining homework assignments.

If you have been identified as a Student Disability Access Center (SDAC) student, please let the Center know you are taking this class. If you suspect you should be an SDAC student, please schedule an appointment with them for an evaluation. I happily and discretely provide the recommended accommodations for those students identified by the SDAC. Please contact your instructor one week before an exam so we can make appropriate accommodations. Website: https://www.studenthealth.virginia.edu/sdac

If you are affected by a situation that falls within issues addressed by the SDAC and the instructor and staff are not informed about this in advance, this prevents us from helping during the semester, and it is unfair to request special considerations at the end of the term or a?er work is completed. So we request you inform the instructor as early in the term as possible your circumstances. If you have other special circumstances (athletics, other university-related activities, etc.) please contact your instructor and/or TA as soon as you know these may affect you in class.

2 Schedule

Week	Day	Date	Topic	Assessments
01	Mon	07/17	M01 Getting Started	
02	Tue	07/18	M02 Introducing Python	
03	Wed	07/19	M03 Control Structures	
04	Thu	07/20	M04 Functions	
05	Fri	07/21	Review Day	
06	Mon	07/24	M05 NumPy	
07	Tue	07/25	M06 Pandas	
08	Wed	07/26	M06 Pandas, continued	
09	Thu	07/27	M07 Classes	
10	Fri	07/28	M08 Testing	
11	Mon	07/31	M09 Modules and Packages	
12	Tue	08/01	M10 Basic R	
13	Wed	08/02	M11 Dplyr	
14	Thu	08/03	Final Projects	
15	Fri	08/04	Review Day	
16	Mon	08/07	M12 Visualization	
17	Tue	08/08	M13 Agile	
18	Wed	08/09	Projects	
19	Thu	08/10	Projects	
20	Fri	08/11	Final Exam	

3 Final Project

DS 5100 | Summer 2023 | Residential

4 Overview

For your Final Project, you will write, test, use, package, and publish a Python module and accompanying files.

The module will implement a simple Monte Carlo simulator using a set of three related classes — a Die class, a Game class, and an Analyzer class.

The classes are related in the following way: Game objects are initialized with a Die object, and Analyzer objects are initialized with a Game object.

B[Game] –> C[Analyzer]

```
In this simulator, a "die" can be any discrete random variable
associated with a stochastic process, such as using a deck of card,
flipping a coin, rolling an actual die, or speaking a language.

The project is designed to integrate what you have learned in this class
by calling upon the following areas of knowledge:

-   Basic syntax, expressions, and statements in Python.
-   Python Classes with initialization methods.
-   Data manipulation with NumPy and Pandas.
-   Literate programming with docstrings and documentation.
-   Unit testing with Unittest.
-   Simple plotting with Pandas.
-   Program modularization and packaging with Setuptools.
-   GitHub for managing and sharing code.

# Class Definitions

The following class definitions provide blueprints for creating the
classes for your simulator. Note that although these provide some
specific instructions, some elements of your code are left to your
interpretation.

## The Die class
```

```
### **General Definition**

-   A die has $N$ sides, or "faces", and $W$ weights, and can be rolled
    to select a face.

    -   For example, a "die" with $N = 2$ is a coin, and a one with
        $N = 6$ is a standard die.

    -   Normally, dice and coins are "fair," meaning that the each side
        has an equal weight. An unfair die is one where the weights are
        unequal.

-   Each side contains a unique symbol. Symbols may be all alphabetic or
    all numeric.

-   $W$ defaults to $1.0$ for each face but can be changed after the
    object is created.

-   The weights are just numbers, not a normalized probability
    distribution.

-   The die has one behavior, which is to be rolled one or more times.

### **Specific Methods and Attributes**

**An initializer**.

-   Takes a NumPy array of faces as an argument. Throws a `TypeError` if
    not a NumPy array.

-   The array's data type (`dtype`) may be strings or numbers.

-   The array's values must be distinct. Tests to see if the values are
    distinct and raises a `ValueError` if not.

-   Internally initializes the weights to $1.0$ for each face.

-   Saves both faces and weights in a private data frame with faces in
    the index.

**A method to change the weight of a single side.**
```

```
-   Takes two arguments: the face value to be changed and the new
    weight.

-   Checks to see if the face passed is valid value, i.e. if it is in
    the die array. If not, raises an `IndexError`.

-   Checks to see if the weight is a valid type, i.e. if it is numeric
    (integer or float) or castable as numeric. If not, raises a
    `TypeError`.

**A method to roll the die one or more times.**

-   Takes a parameter of how many times the die is to be rolled;
    defaults to $1$.

-   This is essentially a random sample with replacement, from the
    private die data frame, that applies the weights.

-   Returns a Python list of outcomes.

-   Does not store internally these results.

**A method to show the die's current state.**

-   Returns a copy of the private die data frame.

## The Game class

### General Definition

-   A game consists of rolling of one or more similar dice (Die objects)
    one or more times.

-   By similar dice, we mean that each die in a given game has the same
    number of sides and associated faces, but each die object may have
    its own weights.

-   Each game is initialized with a Python list that contains one or
    more dice.

-   Game objects have a behavior to play a game, i.e. to roll all of the
```

```
    dice a given number of times.

-   Game objects only keep the results of their most recent play.

### Specific Methods and Attributes

**An initializer**.

-   Takes a single parameter, a list of already instantiated similar
    dice.

-   Ideally this would check if the list actually contains Die objects
    and that they all have the same faces, but this is not required for
    this project.

**A play method.**

-   Takes an integer parameter to specify how many times the dice should
    be rolled.

-   Saves the result of the play to a private data frame.

-   The data frame should be in wide format, i.e. have the roll number
    as a named index, columns for each die number (using its list index
    as the column name), and the face rolled in that instance in each
    cell.

**A method to show the user the results of the most recent play.**

-   This method just returns a copy of the private play data frame to
    the user.

-   Takes a parameter to return the data frame in narrow or wide form
    which defaults to wide form.

-   The narrow form will have a `MultiIndex`, comprising the roll number
    and the die number (in that order), and a single column with the
    outcomes (i.e. the face rolled).

-   This method should raise a `ValueError` if the user passes an
    invalid option for narrow or wide.
```

```
## The Analyzer class

### General Definition

An Analyzer object takes the results of a single game and computes
various descriptive statistical properties about it.

### Specific Methods and Attributes

**An initializer**.

-   Takes a game object as its input parameter. Throw a `ValueError` if
    the passed value is not a Game object.

**A jackpot method.**

-   A jackpot is a result in which all faces are the same, e.g. all ones
    for a six-sided die.

-   Computes how many times the game resulted in a jackpot.

-   Returns an integer for the number of jackpots.

**A face counts per roll method.**

-   Computes how many times a given face is rolled in each event.

    -   For example, if a roll of five dice has all sixes, then the
        counts for this roll would be $5$ for the face value '6' and $0$
        for the other faces.

-   Returns a data frame of results.

-   The data frame has an index of the roll number, face values as
    columns, and count values in the cells (i.e. it is in wide format)..

**A combo count method.**

-   Computes the distinct combinations of faces rolled, along with their
    counts.
```

```
-   Combinations are order-independent and may contain repetitions.

-   Returns a data frame of results.

-   The data frame should have an MultiIndex of distinct combinations
    and a column for the associated counts.

**An permutation count method.**

-   Computes the distinct permutations of faces rolled, along with their
    counts.

-   Permutations are order-dependent and may contain repetitions.

-   Returns a data frame of results.

-   The data frame should have an MultiIndex of distinct permutations
    and a column for the associated counts.

## General Requirements for Classes

-   All classes and methods must have appropriate docstrings.

-   Class docstrings should describe the general purpose of the class.

-   Method docstrings should describe the purpose of the method, any
    input arguments, any return values if applicable, and any changes to
    the object's state that the user should know about.

-   Input argument descriptions should describe data types and formats
    as well as any default values.

-   You may use language included in this document to create these
    docstrings.

# Unit Tests

Write a unit test file using the Unittest package containing **at least
one method for each method in each of the three classes** above. As a
general rule, each test method should verify that the target method
creates an appropriate data structure.
```

```
# Scenarios

To demonstrate the use of your simulator, you will produce a Jupyter
Notebook that performs the following scenarios, each consisting of a set
of tasks:

## Scenario 1: A 2-headed Coin

1.  Create a fair coin (with faces $H$ and $T$) and one unfair coin in
    which one of the faces has a weight of $5$ and the others $1$.

2.  Play a game of $1000$ flips with two fair dice.

3.  Play another game (using a new Game object) of $1000$ flips, this
    time using two unfair dice and one fair die. For the second unfair
    die, you can use the same die object twice in the list of dice you
    pass to the Game object.

4.  For each game, use an Analyzer object to determine the raw frequency
    of jackpots --- i.e. getting either all $H$s or all $T$s.

5.  For each analyzer, compute relative frequency as the number of
    jackpots over the total number of rolls.

6.  Show your results, comparing the two relative frequencies, in a
    simple bar chart.

## Scenario 2: A 6-sided Die

1.  Create three dice, each with six sides having the faces $1$ through
    $6$.

2.  Convert one die to an unfair one by weighting the face $6$ five
    times more than the other weights (i.e. it has weight of $5$ and the
    others a weight of $1$ each).

3.  Convert another die to be unfair by weighting the face $1$ five
    times more than the others.

4.  Play a game of $10000$ rolls with $5$ fair dice.
```

```
5.  Play a game of $10000$ rolls with $2$ unfair dice, one as defined in
    steps #2 and #3 respectively, and $3$ fair dice.

6.  For each game, use an Analyzer object to determine the relative
    frequency of jackpots and show your results, comparing the two
    relative frequencies, in a simple bar chart.

7.  ~~Also compute~~ $10$ ~~most frequent combinations of faces for each
    game.~~

8.  ~~Plot each of these combination results as bar charts.~~

## Scenario 3: Letters of the Alphabet

1.  Create a "die" of letters from $A$ to $Z$ with weights based on
    their frequency of usage as found in the data file
    `english_letters.txt`. Use the frequencies (i.e. raw counts) as
    weights.

2.  Play a game involving $4$ of these dice with $1000$ rolls.

3.  Determine wow many distinct permutations in your results are actual
    English words, based on the vocabulary found in
    `scrabble_words.txt`.

4.  Repeat steps #2 and #3 using $5$ dice and compare the results. Which
    word length generates a higher percentage of English words?

# Submission

Details for submission will be provided in Canvas.

# About the README file

The `README.md` file will be your the main source of documentation for
your users, in addition to your use of docstrings in your code. The file
should consist of the following sections:

-   **Metadata**: Specify your name and the project name (i.e. Monte
    Carlo Simulator).
```

```
-   **Synopsis:** Show brief demo code of how the classes are used, i.e.
    code snippets showing how to install, import, and use the code
    to (1) create dice, (2) play a game, and (3) analyze a game. You can
    use preformatted blocks for the code.

-   **API description**: A list of all classes with their public methods
    and attributes. Each item should show their docstrings. All
    parameters (with data types and defaults) should be described. All
    return values should be described. Do not describe private methods
    and attributes.

# Collaboration

You may work in groups to discuss idea and approaches, but all
deliverables must be yours.

# Deliverables

To complete the project, you will create the following deliverables:

\_\_ A `Die` Class.

\_\_ A `Game` Class.

\_\_ An `Analyzer` Class.

\_\_ A module file named `montecarlo.py` that contains the above three
classes.

\_\_ Unit tests for each method in each class stored in a file named
`montecarlo_test.py`.

\_\_ A text file containing the succsessful results of running the
previous test file named `montecarlo_test_results.txt`.

\_\_ A scenario script that instantiates the classes in a Jupyter
notebook called `montecarlo_demo.ipynb`. NOTE: This file can be the
notebook template provided.

\_\_ `docstring`s for each class and method (including the
initializers).
```

~~~
\_\_ The appropriate files and directories in a project directory to
distribute the above code, including a package subdirectoy called
`montecarlo` with the module file inside of it.

\_\_ A private GitHub repo called `<user_id>_ds5100_montecarlo` that
contains the package and is shared with the professor and grader.

\_\_ A `README.md` file that describes for a new user the purpose of the
package and how to use it.

\_\_ A license file, such as the MIT license.

\_\_ A `.gitignore` file for Python projects.

\_\_ Everything named properly when names are specified.


`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Im5vdGVib29rcy9NMDBfRmluYWxQcm9qZWN0In0= -

```{=html}
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Im5vdGVib29rcy9NMDBfRmluYWxQcm9qZWN0IiwiYm9
~~~

5 Final Project Report

- Class: DS 5100
- Student Name:
- Student Net ID:
- This URL: a URL to the notebook source of this document

6 Instructions

Follow the instructions in the Final Project isntructions and put your work in this notebook.

Total points for each subsection under **Deliverables** and **Scenarios** are given in parentheses.

Breakdowns of points within subsections are specified within subsection instructions as bulleted lists.

This project is worth **50 points**.

7 Deliverables

7.1 The Monte Carlo Module (10)

- URL included, appropriately named (1).
- Includes all three specified classes (3).
- Includes at least all 12 specified methods (6; .5 each).

Put the URL to your GitHub repo here.

Repo URL:

Paste a copyy of your module here.

NOTE: Paste as text, not as code. Use triple backticks to wrap your code blocks.

```
# A code block with your classes.
```

7.2 Unitest Module (2)

Paste a copy of your test module below.

NOTE: Paste as text, not as code. Use triple backticks to wrap your code blocks.

- All methods have at least one test method (1).
- Each method employs one of Unittest's Assert methods (1).

```
# A code block with your test code.
```

7.3 Unittest Results (3)

Put a copy of the results of running your tests from the command line here.

Again, paste as text using triple backticks.

- All 12 specified methods return OK (3; .25 each).

8 A text block with the output of a successful test.

8.1 Import (1)

Import your module here. This import should refer to the code in your package directory.

- Module successuflly imported (1).

```
# e.g. import montecarlo.montecarlo
```

8.2 Help Docs (4)

Show your docstring documentation by applying `help()` to your imported module.

- All methods have a docstring (3; .25 each).
- All classes have a docstring (1; .33 each).

```
# help(montecarlo)
```

8.3 `README.md` File (3)

Provide link to the README.md file of your project's repo.

- Metadata section or info present (1).
- Synopsis section showing how each class is called (1). (All must be included.)
- API section listing all classes and methods (1). (All must be included.)

URL:

8.4 Successful installation (2)

Put a screenshot or paste a copy of a terminal session where you successfully install your module with pip.

If pasting text, use a preformatted text block to show the results.

- Installed with `pip` (1).
- Successfully installed message appears (1).

9 Pasted code

10 Scenarios

Use code blocks to perform the tasks for each scenario.

Be sure the outputs are visible before submitting.

10.1 Scenario 1: A 2-headed Coin (9)

Task 1. Create a fair coin (with faces H and T) and one unfair coin in which one of the faces has a weight of 5 and the others 1.

- Fair coin created (1).
- Unfair coin created with weight as specified (1).

Task 2. Play a game of 1000 flips with two fair dice.

- Play method called correclty and without error (1).

Task 3. Play another game (using a new Game object) of 1000 flips, this time using two unfair dice and one fair die. For the second unfair die, you can use the same die object twice in the list of dice you pass to the Game object.

- New game object created (1).
- Play method called correclty and without error (1).

Task 4. For each game, use an Analyzer object to determine the raw frequency of jackpots — i.e. getting either all Hs or all Ts.

- Analyzer objecs instantiated for both games (1).
- Raw frequencies reported for both (1).

Task 5. For each analyzer, compute relative frequency as the number of jackpots over the total number of rolls.

- Both relative frequencies computed (1).

Task 6. Show your results, comparing the two relative frequencies, in a simple bar chart.

- Bar chart plotted and correct (1).

10.2 Scenario 2: A 6-sided Die (9)

Task 1. Create three dice, each with six sides having the faces 1 through 6.

- Three die objects created (1).

Task 2. Convert one of the dice to an unfair one by weighting the face 6 five times more than the other weights (i.e. it has weight of 5 and the others a weight of 1 each).

- Unfair die created with proper call to weight change method (1).

Task 3. Convert another of the dice to be unfair by weighting the face 1 five times more than the others.

- Unfair die created with proper call to weight change method (1).

Task 4. Play a game of 10000 rolls with 5 fair dice.

- Game class properly instantiated (1).
- Play method called properly (1).

Task 5. Play another game of 10000 rolls, this time with 2 unfair dice, one as defined in steps #2 and #3 respectively, and 3 fair dice.

- Game class properly instantiated (1).
- Play method called properly (1).

Task 6. For each game, use an Analyzer object to determine the relative frequency of jackpots and show your results, comparing the two relative frequencies, in a simple bar chart.

- Jackpot methods called (1).
- Graph produced (1).

10.3 Scenario 3: Letters of the Alphabet (7)

Task 1. Create a "die" of letters from A to Z with weights based on their frequency of usage as found in the data file `english_letters.txt`. Use the frequencies (i.e. raw counts) as weights.

- Die correctly instantiated with source file data (1).
- Weights properly applied using weight setting method (1).

Task 2. Play a game involving 4 of these dice with 1000 rolls.

- Game play method properly called (1).

Task 3. Determine how many permutations in your results are actual English words, based on the vocabulary found in `scrabble_words.txt`.

- Use permutation method (1).
- Get count as difference between permutations and vocabulary (1).

Task 4. Repeat steps #2 and #3, this time with 5 dice. How many actual words does this produce? Which produces more?

- Successfully repreats steps (1).
- Identifies parameter with most found words (1).

11 Submission

When finished completing the above tasks, save this file to your local repo (and within your project), and them push it to your GitHub repo.

Then convert this file to a PDF and submit it to GradeScope according to the assignment instructions in Canvas.

Part II

M01 Getting Started

Topics

- Introduce the course
- Access Rivanna
- Explore the Unix command line
- Explore use of Git and GitHub

Outcomes

- Become familiar with UVA's compute resources Rivanna
- Become familiar with the command line, e.g. bash
- Know the difference between Git and GitHub
- Know how to fork and clone a repository for personal use
- Know how to push content to a repository that you own
- Know how to make a pull request to a repository that you don't own

12 About Rivanna

12.1 Introduction

A useful infrastructural resource for this course is Rivanna, UVA's high-performance computing (HPC) cluster. Each student has an account on Rivanna and access to resources there based on participation in this course. We will use Rivanna in our class for both Python and R.

This page describes some of the tools available for your use in this course. For information about Rivanna, see this introduction. Resources for getting help, including a knowledge base and ticket system, are found at the Support Option's Page on UVA's Research Computing website.

You may need to use VPN to access Rivanna from an off-grounds location. To install VPN on your computer, go to the ITS VPN page for instructions. Note that you should connect to "UVA Anywhere," not to any of the higher security options. Course Allocation

This course has been allocated compute and space resources on Rivanna. The names of the resources are given below. The allocation ID needs to be entered to access certain tools. The storage path is accessible to you on the remote server.

- Allocation ID: `msds_ds5100`
- Storage path: `/project/MSDS_DS5100` (Don't use unless directed to.)

12.2 Tools

UVA Research Computing provides you with a suite of tools to access Rivanna. These tools are accessible through the menu on the UVA OpenOnDemand Dashboard page. Below are some brief descriptions of the tools.

File Explorer. A web-based GUI to access the file system of the remote server. Can be used to create, move, and delete directories and files, and to edit the contents of files (see Editor). You can also upload and download files through this interface. The File Explorer is useful to view your remote content and manage files and directories without having to use the command line. Note that not all operations can be performed through this interface.

Find under "Files" in the menu.

Editor. A web-based text editor launched from the File Explorer to view and edit text files on the remote server. Although not as sophisticated as VS Code (below), this is very useful for editing data and code files without having to use a command line editor. One advantage over VS Code is that it does not need to be launched – which means it does not time out like the Interactive Apps listed below.

The Editor is launched from the File Explorer.

SSH Shell Access (Terminal). Access to the command line of the remote server. Use this to open a terminal window to perform Linux commands directly. Note that It is necessary to use a terminal to install and run certain programs on the remote server.

Find under "Clusters" in the menu.

You can also access the remote command line via SSH on your local computer. Just enter the following on the command line of either a PC or Mac:

```
ssh -Y <userid>@hpc.rivanna.virginia.edu
```

Replace `<userid>` with your UVA Net ID, e.g. `abc2x`.

Be suer to be running VPN if you are accessing Rivanna from an off-grounds location.

12.3 Interactive Apps

These tools must be launched by specifying a set of parameters, including the allocation you are using. They are also timed and will close when time is up. Be sure to give yourself enough time when launching these, and to be aware of how much time you have when working.

Note also that you should allocate the fewest resources necessary to do the work you plan to do. This saves resources on the remote host, but also allows your app to launch more quickly. If you ask for an excessive amount of resources, you may wait a long time (e.g. hours) to have your app launched.

Desktop. Access to a GUI desktop to the remote server. This provides a access to various applications on the server, including a web browser, a file explorer, and terminal windows. Using this is not necessary if you can get by with the tools listed above.

Find under "Interactive Apps > Desktop" in the menu.

VS Code. Access to Visual Studio Code on the remote server. This is a fully functional instance of the IDE.

Find under "Interactive Apps > Code Server" in the menu.

Jupyter. Access to Jupyter Lab on the remote server. Find under "Interactive Apps > Jupyter Lab" in the menu.

RStudio. Access to Jupyter Lab on the remote server.

Find under "Interactive Apps > RStudio Server" in the menu.

12.4 For More Information

UVA's Research Computing unit provides resources for learning how to use Rivanna. Here are two slide decks that you may find useful:

- Introduction to Rivanna
- Using Rivanna from the Command Line

13 Using Unix

13.1 Introduction

The Unix family of operating systems provide users with a command line interface (CLI) to execute commands and get things done. They also, typically provide GUIs but we won't go into those here.

The Unix family includes all varieties of Linux and the Mac OS (which is based on FreeBSD).

The command line that you actually interact with – the set of commands available to you – is called a shell, and there are several shells that you can run on your system. The most typical shell in use today is called bash which stands for Bourne Again Shell, since it is an improved version of bsh (The Bourne Shell). New versions of MacOS use the Z shell (zsh). The commands in these two shells are mostly similar, but there are subtle differences.

Windows has shells too for its command line interface. The default shell is DOS, but is also has PowerShell as an advanced (and very capable) option.

For more information, check out these resources:

- UVA Research Computing's Unix tutorial.
- Newham, 2005, Learning the bash Shell, O'Reilly Media.
- Jeroen Janssens, 2021, Data Science From the Command Line, O'Reilly Media.
- Neal Stephenson, 1999, In the Beginning Was The Command Line. (PDF version.)

13.2 Basic Commands

In this course, you don't need to know very many Unix shell commands, but you should be comfortable working from the command line to perform basic tasks. This is because some things can only be performed from the command line, such as installing some essential software. Here is a list of basic commands.

Navigating filesystems and managing directories:

- `cd` – change directory
- `pwd` – show the current directory
- `ln` – make links and symlinks to files and directories

- `mkdir` – make new directory
- `rmdir` – remove directories in Unix

Navigating filesystem and managing files and access permissions:

- `ls` – list files and directories
- `cp` – copy files (work in progress)
- `rm` – remove files and directories (work in progress)
- `mv` – rename or move files and directories to another location
- `chmod` – change file/directory access permissions
- `chown` – change file/directory ownership

13.3 Text file commands

Most of important configuration in Unix is in plain text files, these commands will let you quickly inspect files or view logs:

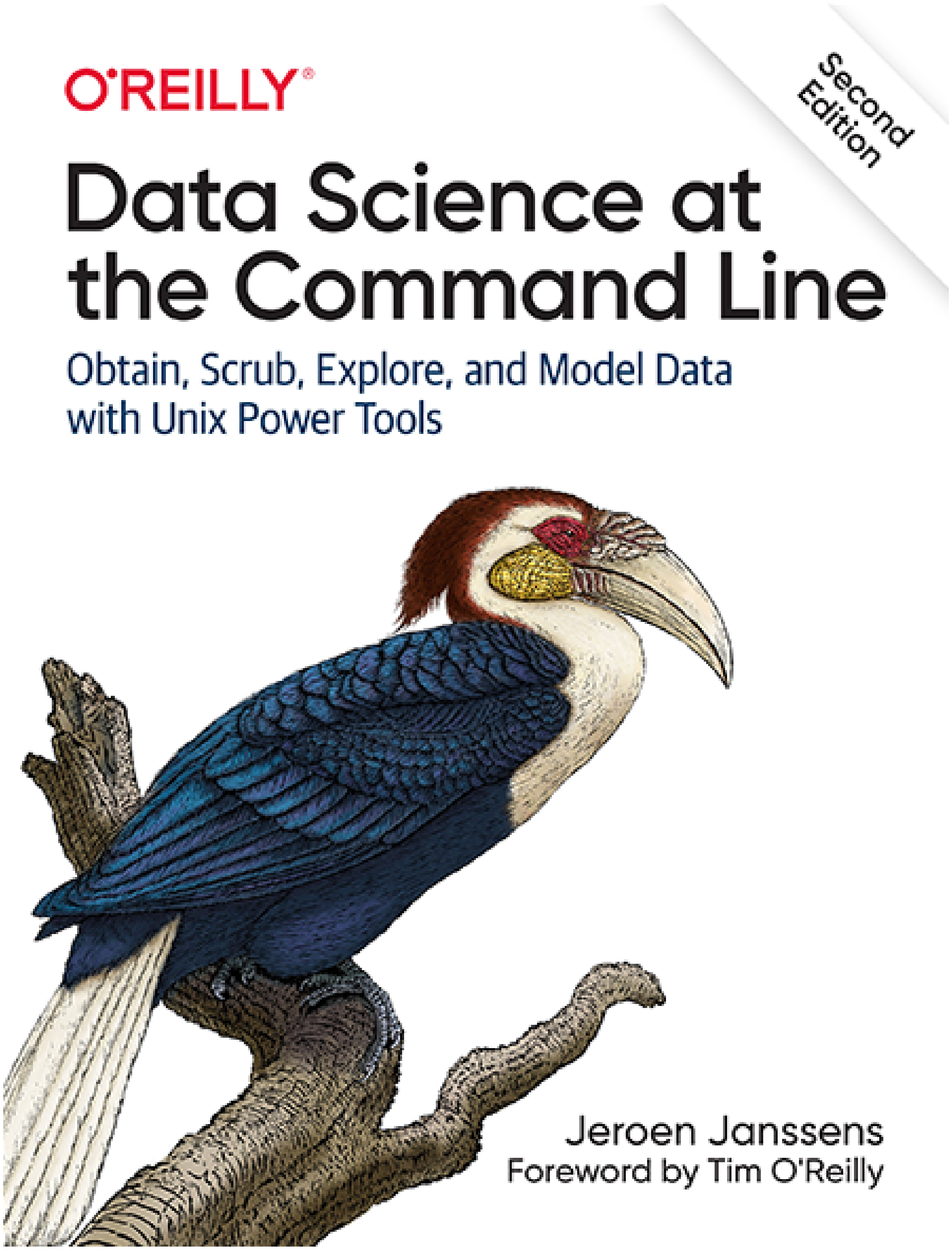
- `cat` – concatenate files and show contents to the standard output
- `more` – basic pagination when viewing text files or parsing Unix commands output
- `less` – an improved pagination tool for viewing text files (better than more command)
- `head` – show the first 10 lines of text file (you can specify any number of lines)
- `tail` – show the last 10 lines of text file (any number can be specified)
- `grep` – search for patterns in text files

13.4 Miscellaneous

- `clear` – clear screen
- `history` – show history of previous commands

13.5 Command Line Cool

Although we will not be using the command line to this degree, you should know that it is a powerful environment for doing data science work. The book Data Science from the Command Line makes the case for using the command line to perform many tasks that we often perform with more resource intensive (i.e. bloated) tools such as Python and R. At some point in your early DS career, you may want to look at this. The book itself is also a great introduction to data science!


O'REILLY
Second Edition
Data Science at the Command Line
Obtain, Scrub, Explore, and Model Data with Unix Power Tools
Jeroen Janssens
Foreword by Tim O'Reilly

One last thing – for fun you may want to read Neal Stephenson's "In the Beginning Was The Command Line", a kind of cyberpunk history of the topic. Stephenson, by the way, is the author who coined the term "metaverse" in the novel *Snowcrash*.

14 SSH for GitHub

14.1 Overview

- This method will allow you to interact with your repos hosted on GitHub without having to enter your login credentials each time.
- You will create an SSH key on your local machine. By "local machine," we mean the machine where you will be working and pulling to, e.g your laptop or Rivanna. In other words, it's local relative to GitHub.
- SSH keys have a public and private component. These are hash strings that are stored in files. Both will be generated on your machine.
- You will copy and paste the generate public key to your GitHub account.
- Going forward, you will clone from your account using the SSH protocol.

14.2 Steps

14.2.1 Part A

On your local machine

Get to the command line (i.e. the shell).

1. On a Mac, open Terminal.
2. If you are on Windows and you have admin rights, first install `git-bash`. Otherwise follow this tutorial from Microsoft.
3. On Rivanna, either connect via SSH or use Rivanna Shell Access (under Clusters).

Move into your root directory and enter `cd`.

Generate the key.

1. Enter: `ssh-keygen -t ed25519 -C "your_email_id@example.com"`, using your email address.
2. Be sure to use the email address associated with your GitHub account in the above command.

At the prompt, type in a secure passphrase.

- You don't have to do this, but it is advised.
- Create a memorable sentence.
- A good passphrase should have at least 15, preferably 20 characters and be difficult to guess. It should contain upper case letters, lower case letters, digits, and preferably at least one punctuation character.

Add the key to `ssh-agent`.

1. Enter: `eval "$(ssh-agent -s)"`
2. Enter: `ssh-add ~/.ssh/id_ed25519`

If you're using macOS Sierra 10.12.2 or later, you will need to modify your `~/.ssh/config` file to automatically load keys into the ssh-agent and store passphrases in your keychain. Follow the instructions here (at step 2).

14.2.2 Part B

On your GitHub account

Get the public key that was just generated.

1. Enter: `more ~/.ssh/id_ed25519.pub`
2. Copy the result to your clipboard (e.g. by blocking off the line and entering Ctrl-C).
3. The key should begin `ssh-ed25519` and end with the email address you used in generating the key. In between it will have a long string of alphanumeric characters.

On GitHub, go to your account settings and select "SSH and GPG Keys" from the side menu.

- A link to your account settings can be found in the drop-down list produced by clicking on your user icon in the upper right of the website.

Under "SSH keys," press the "New SSH Key" button.

1. Add a brief title describing the context of the key, i.e. the local machine where it was generated, e.g. Rivanna.
2. Choose "Authentication Key" as the Key type.
3. Paste the key into the Key text area.
4. Submit the form by pressing "Add SSH key."

You are now good to go. Whenever you clone a site from your GitHub account, choose the SSH link.

14.3 Information Sources

The GitHub site has lots of excellent documentation. Here are some pages you may find useful.

- About SSH
- Generate the key
- About pass phrases
- Adding the key to GitHub
- Updating repos with SSH

15 Git and GitHub

15.1 Introduction

Git and GitHub are two tools that work together, but it is important to understand what each does and how they are different to each other.

Here are some basic things to know:

1. Git is a stand-alone version control system that runs on a variety of platforms, including Linux, MacOS, and Windows.
2. GitHub is a company that offers a cloud-based Git repository hosting service that makes it easy to use Git for version control, collaboration, and sharing code. This service is offered through a website.
3. There are other Git hosting services out there, including GitLab, and open source tool that can be installed on a local server.
4. GitHub builds on top of Git and adds some functionality to it, while Git can interact with GitHub through actions like cloning, pushing, and pulling. However, Git does not require GitHub to function.
5. Git does not have a fork command. GitHub (and other hosting services such as GitLab) have added this command as a convenient way to copy repositories.

15.2 Using Git and GitHub Together

Source

A basic series of actions one continually makes with Git are the following:

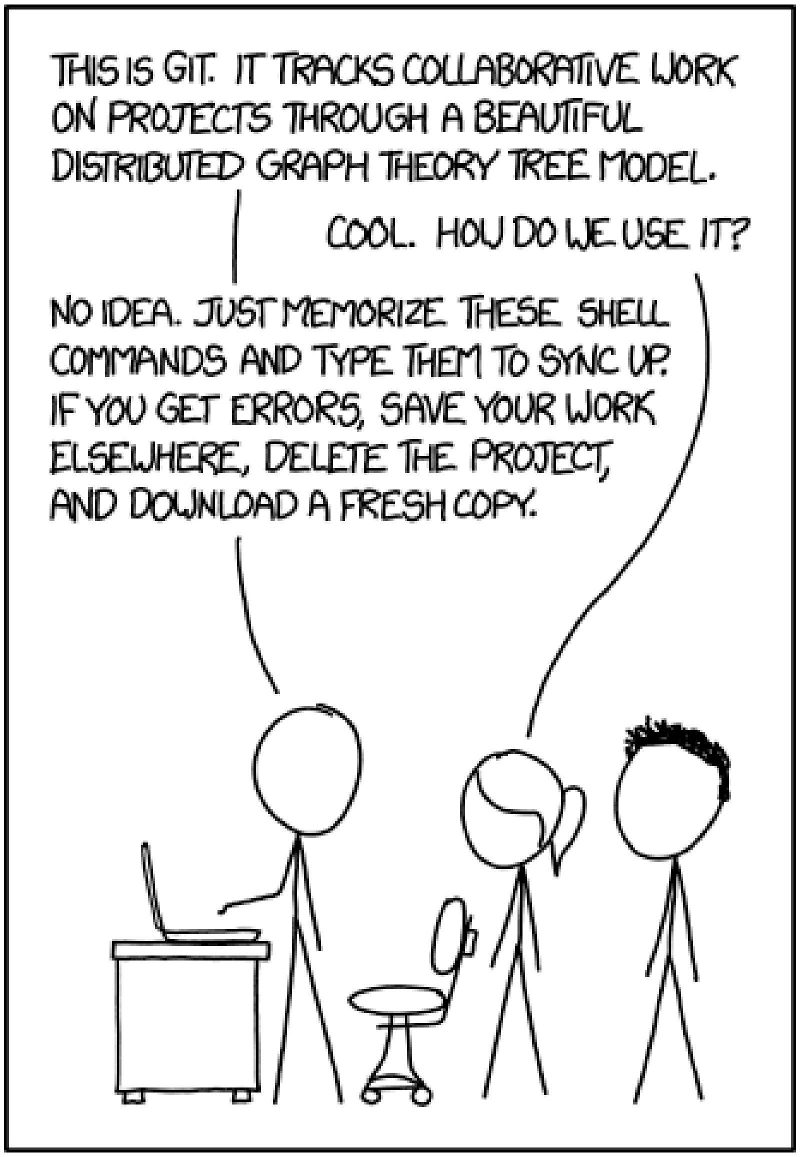

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Figure 15.1: XKCD #1597

Action	Description	Command
Fork	Forking a repo makes a copy of someone else's repo in your GitHub account, which you can now modify. Note: Forking does not have to be performed if you are not interested in altering the code. You can clone it directlty.	This is done through GitHub's web interface by clicking on the Fork button.
Clone	Cloning the repo to a workspace to which you have access, whether a local machine or a remote resource (e.g. Rivanna).	`git clone <repo>`
Add	Creating or editing a file locally and then adding it to the list of files git will keep track of.	`git add <filename>`Note that you may use wildcards here, e.g. *, to add more than one file at a time.
Commit	Committing the changes made to the file by adding them to the repo's database of changes. This is accompanied by a brief, descriptive message of the changes made.	`git commit -m "What you did"`
Push	Pushing the changes to the remote repo that you cloned from. This uploads both the files and the database changes to the repo.	`git push`
Pull	Pulling, i.e. downloading, any changes that have been made to the remote repo to your local repo. This usually happens if you are working in teams on the same project.	`git pull`
Pull Request	This is a request made to the owner of the original repo to pull in the changes you've made to your forked copy.	This is done through GitHub's web interface. Read the docs.

Action	Description	Command
Fetch Upstream	This refreshes the content of your forked repo with the content from the original repo.	This is done through GitHub's web interface. Read the docs.

> **Note**
>
> This is not the only pattern to use with Git. Here is another — the Git Fork-Branch-Pull Workflow

Here is a visualization of the process:

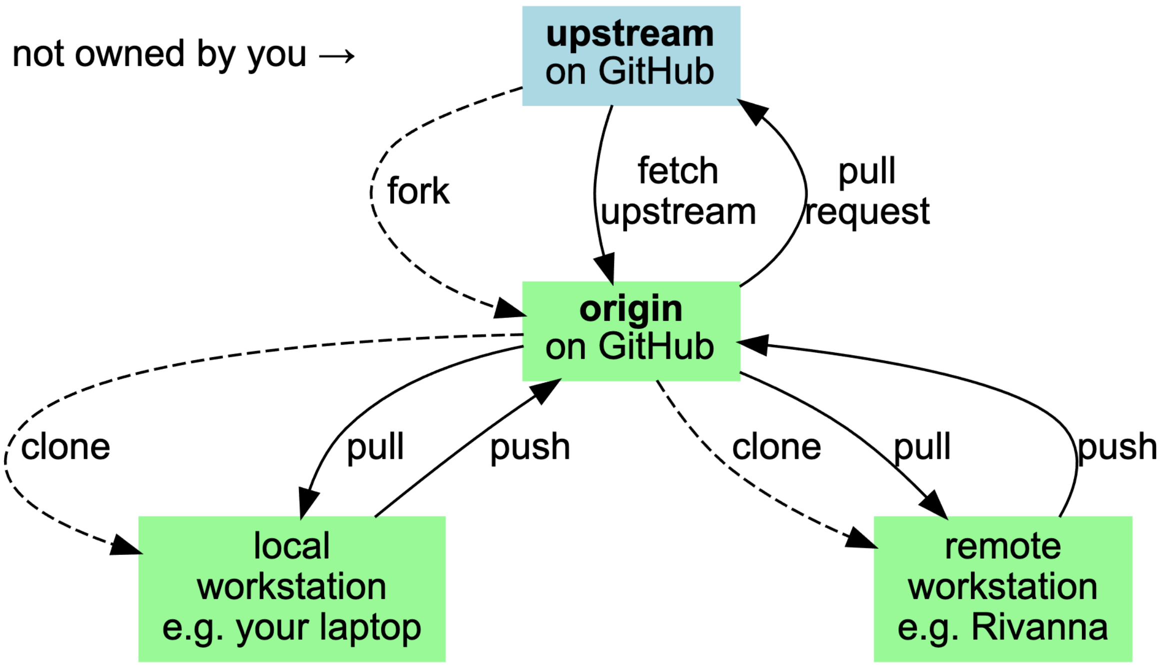


Figure 15.2: Diagram of common git workflow

In this diagram, the dashed lines refer to actions performed only once for a give repo. Forking and cloning are done to acquire a repo, while fetch upstream (aka sync fork) and pull requests on the GitHub server, and pull/push on your local machine, are done repeatedly as you develop and share code.

Note also that the here “Remote workstation” may be confusing; it means remote relative to your laptop, e.g. Rivanna, which we sometimes call local relative to the GitHub repo. In any case, note that these two copies of the same repo do not communicate with each other directly, but rather through their common relationship with the GitHub hosted instance of the repo.

15.3 To Learn More

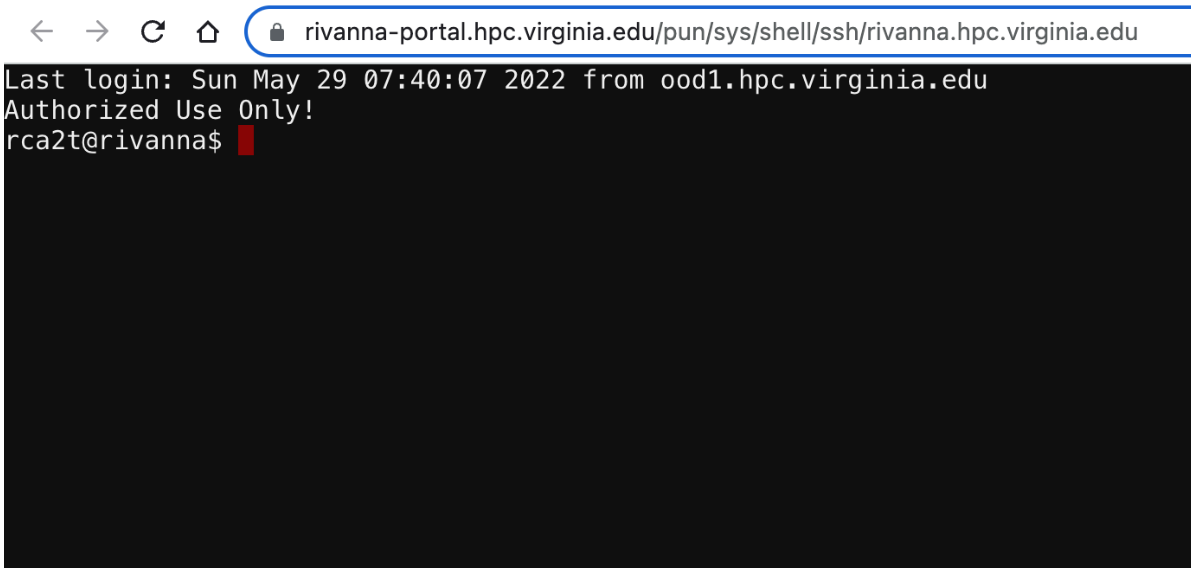
- Videos
- Book

16 Activity: Using Rivanna

After reading the previous documents on Rivanna and Unix, try this activity to get acquainted with the Rivanna high-performance computing cluster at UVA.

To get started, go to OpenOnDemand Dashboard page and from the main menu select Clusters → Rivanna Shell Access.

This should open a terminal to what is called the "shell" of the operating system.


rivanna-portal.hpc.virginia.edu/pun/sys/shell/ssh/rivanna.hpc.virginia.edu

```
Last login: Sun May 29 07:40:07 2022 from ood1.hpc.virginia.edu
Authorized Use Only!
rca2t@rivanna$ 
```

Figure 16.1: Screenshot of Rivanna shell

Rivanna uses Linux, a member of the Unix family of operating systems. Many cloud resources use Linux.

Understanding how to do work from the command line on such systems is an essential skill of the data scientist.

If you have never used the command line, have no fear! Just enter the commands exactly as shown and ask questions in the Teams chat if you are stuck.

Now, create a directory for your course and this course by entering the following commands:

```
cd Documents
mdkir MSDS
cd MSDS
mkdir DS5100
cd DS5100
```

If the `Documents` directory does not exist, create that first using the mkdir command.

- `cd` means "change directory," and is a basic Unix command.
- `mkdir` means "make directory." It's also a basic Unix command.

Note that you can use the tab key to complete path and command names as you type.

You don't have to, but it would be a good idea to create subdirectories for any of your courses that use Rivanna.

More information about Unix shell commands can be found the document Unix Shell Commands.

17 Activity: Using Git and GitHub

In this activity, you will go through steps of using Git and GitHub covered in the the reading on GitHub. You can also draw on the what you learned in the Technical Orientation. At this point, you also should be able check off the following items:

- Understand the difference between Git and GitHub.
- Understand the purpose of Git and Github for data science work.
- Ensure Git is installed on your computer.
- Understand how to find a repository on GitHub.

Let's apply and extend this knowledge now with our course repo.

Be sure you are inside the course directory on Rivanna we created earlier.

We assume you have already created a GitHub account.

Also, before you get started, follow these instructions to set an SSH key. You can create this on both your computer and Rivanna, but for the assignment you need only create it on Rivanna.

***Fork* the course GitHub hosted repository ("repo") to your GitHub account.**

Go to https://github.com/ontoligent/DS5100-2023-07-R in your web browser.

> Note
>
> This is the course repo — all of the course notebooks and other code will be available here. Each week, you will access your course materials here.

Click on the Fork icon in the upper right and follow the prompts to finish the process.

You should end up at the web page of your newly forked repo.

You will now have a copy of the repo in your GitHub account.

***Clone* the forked repo for this course inside of your course directory on Rivanna.**

Find the green Code button and click on it. You should see something like this:

Make sure you have selected the SSH option.

❗ Important

Note: This requires that you have SSH set up.

Then click on the copy icon and paste the value into the following command:

```
git clone https://github.com:<github_user_name>/DS5100-2023-07-R.git
```

❗ Important

Be sure to clone the repo from *your* GitHub account, replacing `<github_user_name>` with your GitHub user name. Do not just cut-and-paste the line above!

You now have a copy the course repo to your account on Rivanna.

This will be the directory you created in your pre-class activities under Documents/.

***Create* a new file in your newly cloned repo.**

Go to your command line window on Rivanna.

Use `cd` to move into the directory just created by the clone operation.

Move into the directory `notebooks/M01_GettingStarted/hello`

❗ Important

Make sure you are in this directory before proceeding.

If you get lost – for example if you moved around the file system before this step – you can cd to the absolute path:

```
cd ~/Documents/MSDS/DS5100/DS5100-2023-07-R/notebooks/M01_GettingStarted/hello
```

Note that the tilde sign ~ stands for the path to your home directory.

Using the file editor on Rivanna, create and save new file called `<userid>_hello.txt`, replacing `<userid>` with your actual user ID, e.g. `rca2t_hello.txt`.

In the file, introduce yourself by answering the question: What is the most recent film you watched and enjoyed?

Save the file.

***Add* and *commit* the changes you made.**

Now do the following:

```
git add <userid>_hello.txt
git commit -m "Created file for class"
```

***Push* your new file to the repo on GitHub.**

Since you have SSH set up, you can issue the following command without having to enter a password:

```
git push
```

Create a *Pull Request*

Finally, make a pull request to have your file added to the original site. To do this, follow these steps:

Click on the "Pull requests" menu item (see image below) on the web page for your repo.

Figure 17.1: Image of pull request button on GitHub

Click on the green "New pull request" button.

Click on the green "Create pull request" button.

Give the request the title "In-class activity" and then press the green "Create pull request" button at the bottom of the form.

Now the ball is in the instructor's court to merge the request with the original. If you put your file in the right place and named it properly, it will be merged.

17.1 Going Forward

During the semester, you will not be making pull requests to submit your work. We do it here to demonstrate the concept since it is so basic to working with GitHub in the real world.

Instead of making pull requests, you will be using a separate repository for your work So, you will be working with two repositories going forard:

1. **The Course Repo**, which is where you will get course materials. This should be updated each day.
2. **Your Assessments Repo**, which is where you will be your finished work as assigned.

Part III

M02 Introducing Python

Topics

- Running Python code.
- Python's basic data types.
- Python's primary operators associated with each data type.
- Python's built-in data structures.

Outcomes

- Run Python from the command line on Rivanna.
- Create and run a Jupyter Notebook on Rivanna.
- Describe the difference between data from the perspective of data science versus computer science.
- Know the primary data types in Python and their basic operators.
- Know the built-in list-like data structures in Python and the basic methods and functions associated with them.

Readings

Required

Katz and Katz 2019, Section 1, Preparing the Workspace

Lutz, Learning Python, Part I: Getting Started, Chapter 2

Lutz, Learning Python, Part I: Getting Started, Chapter 3

Lutz, Learning Python, Part II: Types and Operations, Chapters 4–9

Optional

Katz and Katz 2019, Section 1, First Steps in Coding - Variables and Data Types

Built-in Types (Official)

Python Data Types (GFG)

Python Operators (W3S)

Immutable vs Mutable Data Types in Python (Medium)

18 Data and Code

18.1 Code should be simple

An important principle for writing effective and intelligible code is that code should be simple — to quote Einstein, as simple as possible but no simpler.

- A contributing factor to code simplicity is how it is related to the data it is designed to process.
- This relationship depends largely on how the data are structured.
- A program is always written with data in mind — what kind of data it is and how it is structured.

18.2 Simplicity of code follows from the structure of data

There is a view among programmers which, although not orthodoxy, is commonplace.

- It is the idea that the complexity of a program — its algorithms — is a function of the quality of the data structure it processes.
- If a data structure is not well designed, algorithms may be excessively complex and hard to understand.
- However if a data structure is well designed, the algorithms that process them are more robust and intelligible.

18.3 Supporting References

Consider these quotes cited in an essay on Data Structures. by Igor Budasov, reproduced here:

Here's a quote from Linus Torvalds in 2006:

> I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful . . . I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his [sic] code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

Which sounds a lot like Eric Raymond's "Rule of Representation" from 2003:

> Fold knowledge into data, so program logic can be stupid and robust.

Which was just his summary of ideas like this one from Rob Pike in 1989:

> Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

Which cites Fred Brooks from 1975:

> **Representation is the Essence of Programming**
>
> Beyond craftsmanship lies invention, and it is here that lean, spare, fast programs are born. Almost always these are the result of strategic breakthrough rather than tactical cleverness. Sometimes the strategic breakthrough will be a new algorithm, such as the Cooley-Tukey Fast Fourier Transform or the substitution of an n log n sort for an n 2 set of comparisons.
>
> **Much more often, strategic breakthrough will come from redoing the representation of the data or tables.** This is where the heart of your program lies. Show me your flowcharts and conceal your tables, and I shall be continued to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

19 Python Object Types

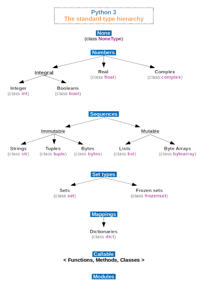
Python is organized into a hierarchy of object types. Sometimes, these are just call **types**.

Objects are the basic unit out of which the language is constructed.

We'll learn about objects later – what they are and how to create your own – but for now just understand that they have two main things associated with them:

- First, they can contain **data**.
- Second, they can have **behaviors**, frequently in relation to the data they contain.

Data types and data structures are kinds of objects.



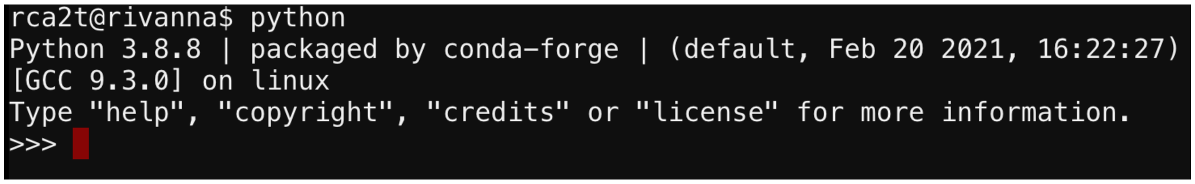
20 Activity: Hello, World!

20.1 The Python Interactive Shell

Log onto the Rivanna shell and move into in the course directory you created for this class.

From the command line, enter python

You should get the Python Shell:



```
rca2t@rivanna$ python
Python 3.8.8 | packaged by conda-forge | (default, Feb 20 2021, 16:22:27)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

This is also called the Python standard REPL, which stands for "Read-Eval-Print Loop".

Make sure you see that you are using version 3 of Python.

If you see Python 2, exit the shell by entering `quit()` and try again by entering python3 at the command line.

At the `>>>` prompt type `print("Hello, World!")` and press return.

If you've never used Python, you've just completed an important ritual. If you have used Python, well, you did it again :-)

20.2 Try `this`

Now, enter following line at the prompt and press return:

```
import this
```

What do you see?

To exit the Python Shell, enter `quit()` or `exit()` and hit return.

20.3 Running Python Files

Now create a file called `hello.py` using the command line editor `nano`. Enter the same commands you used above.

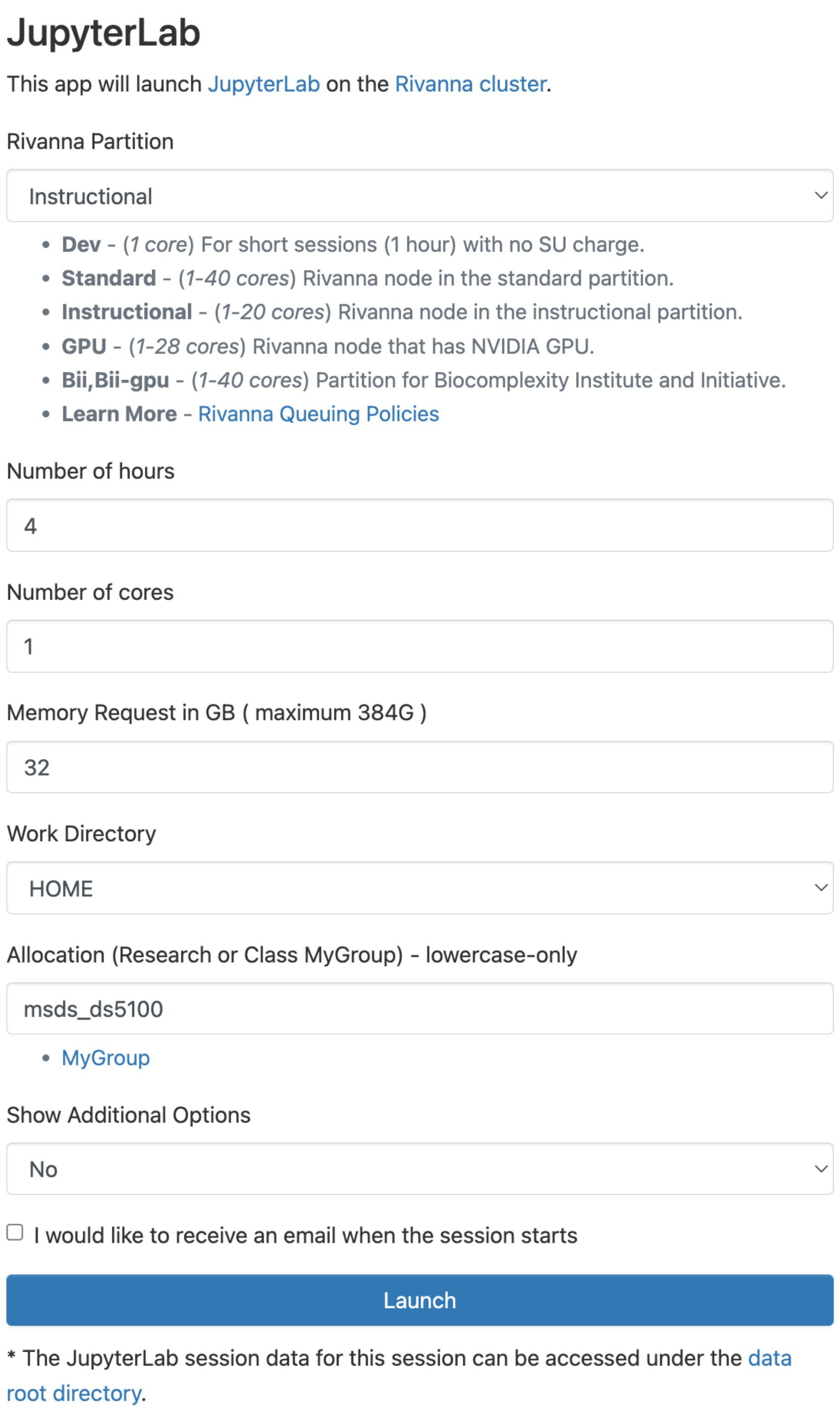
Then run it from the command line by directly invoking the Python interpreter python.

21 Activity: Jupyter Lab

Now that we have run Python on Rivanna from the command line, let's try it using a Jupyter Notebook.

Go the OnDemand site to access Rivanna. As a reminder, the URL is https://rivanna-portal.hpc.virginia.edu/.

From the Interactive Apps menu, select JupyterLab and fill out the form to initiate a new session. Your form should have the following values:



JupyterLab

This app will launch JupyterLab on the Rivanna cluster.

Rivanna Partition

Instructional

- **Dev** - (*1 core*) For short sessions (1 hour) with no SU charge.
- **Standard** - (*1-40 cores*) Rivanna node in the standard partition.
- **Instructional** - (*1-20 cores*) Rivanna node in the instructional partition.
- **GPU** - (*1-28 cores*) Rivanna node that has NVIDIA GPU.
- **Bii,Bii-gpu** - (*1-40 cores*) Partition for Biocomplexity Institute and Initiative.
- **Learn More** - Rivanna Queuing Policies

Number of hours

4

Number of cores

1

Memory Request in GB (maximum 384G)

32

Work Directory

HOME

Allocation (Research or Class MyGroup) - lowercase-only

msds_ds5100

- MyGroup

Show Additional Options

No

☐ I would like to receive an email when the session starts

Launch

* The JupyterLab session data for this session can be accessed under the data root directory.

Tip

Note that you may increase the number of hours, cores, and megabytes of RAM, but asking for too much will increase the time it takes to start your session. So select just the resources needed and enter our course allocation `msds_ds5100` if this value is different than in the image above).

Once the session is ready, launch the notebook.

Once you are in the notebook, use file system tab on the left to get to the directory of your personal assessments repo. Remember, you created two repos for this class — one for course content from the instructor, and one for your own course work. Use the latter for this exercise.

In a code cell in the notebook, enter the code to print `"Hello, World!"`, and run the cell.

Save your notebook as `hello-world.ipynb`.

22 NB: Data Types, Operators, and Expressions

23 Python Data Types

We declare a number of variables with different value types.

By 'type' we mean object type.

Data types and data structures are both types of object.

Data types are created by the way they are written or as keywords …

Here is a series of literal values (called **literals**):

Integers

```
100
```

```
100
```

Floats (decimals)

```
3.14
```

```
3.14
```

```
1, 1.
```

```
(1, 1.0)
```

Strings

Type of quote does not matter, but they must be straight quotes, not "smart quotes" that some word processors use.

Note that there is no explicit **character** type as in Java and other languages.

```
"foo"
```

```
'foo'
```

```
"1"
```

```
'1'
```

```
'foo'
```

```
'foo'
```

Boolean

```
True, False
```

```
(True, False)
```

Nothing

It evaluates to nothing!

```
None
```

```
print(None)
```

```
None
```

Complex

For the physicists and signal processors.

```
5+0j
```

```
(5+0j)
```

24 Getting the type of a value

You can always find out what kind of type you are working with by calling the `type()` function.

```
type(3.14)
type("foo")
type('foo')
type(True)
type(None)
```

```
<class 'float'>
<class 'str'>
<class 'str'>
<class 'bool'>
<class 'NoneType'>
```

25 Assignment

Data are assigned to **variables** using the assignment **operator** `=`.

The variable is always on the **left**, the value assigned to it on the **right**.

This is not the same as mathemtical equality.

Variables are assigned types **dynamically**.

This is in contrast to static typing, where you have define variables by asserting what kind of data values they can hold.

Python figures out what type of data is being set to the variable and implicitly stores that info.

```
integerEx = 8
longIntEx = 22000000000000000000000
floatEx = 2.2
stringEx = "Hello"
booleanEx = True
noneEx = None
```

Note that `type()` returns the type of the value that a variable holds, not the type "variable".

```
type(integerEx)
```

```
<class 'int'>
```

26 Deleting variables with `del()`

```python
x = 101.25
```

```python
x
```

```
101.25
```

```python
del(x)  # delete the variable x
```

```python
x
```

```
NameError: name 'x' is not defined
```

You can't delete values!

```python
del("foo")
```

```
SyntaxError: cannot delete literal (1397139688.py, line 1)
```

27 Get Object Indenity with `id()`

This function returns the identity of an object.

The identity is a number that is guaranteed to be unique and constant for this object during its lifetime (during the program session).

You can think of it as the address of the object in memory.

```
print(id(integerEx))
```

```
4329876032
```

28 Convert Types with Casting Functions

It is possible to convert between types (when it makes sense to do so).

Sometimes conversions are "lossy" – you lose information in the process

28.1 `int()`

```
int?
```

```
Init signature: int(self, /, *args, **kwargs)
Docstring:
int([x]) -> integer
int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
Type:           type
Subclasses:     bool, IntEnum, IntFlag, _NamedIntConstant
```

Float to Int

```
val = 3.8
print(val, type(val))
```

```
3.8 <class 'float'>
```

```
val_int = int(val)
print(val_int, type(val_int))
```

```
3 <class 'int'>
```

String to Float

```
val = '3.8'
print(val, type(val))
```

```
3.8 <class 'str'>
```

```
val_int = float(val)
print(val_int, type(val_int))
```

```
3.8 <class 'float'>
```

Converting string decimal to integer will fail:

```
val = '3.8'
print(val, type(val))
```

```
3.8 <class 'str'>
```

```
val_int = int(val)
print(val_int, type(val_int))
```

```
ValueError: invalid literal for int() with base 10: '3.8'
```

28.2 `ord()`

Converting a character to it's code point

```
ord?
```

```
Signature: ord(c, /)
Docstring: Return the Unicode code point for a one-character string.
Type:      builtin_function_or_method
```

```
ord('a'), ord('A')
```

```
(97, 65)
```

29 Operators

If variables are **nouns**, and values **meanings**, then operators are **verbs**.

In effect, they are **elementary functions** that are expressed in sequential syntax.

`a + b` could have been expressed as `add(a, b)`.

Basically, **each data type is associated with a set of operators** that allow you to manipulate the data in way that makes sense for its type. Numeric data types are subject to mathematical operations, booleans to logical ones, and so forth.

There are also **operations appropriate to structures**. For example, list-like things have membership.

The relationship between types and operators is a microcosm of the relationship betweed data structures and algorithms. **Data structures imply algorithms and algorithms assume data structures.**

The w3schools site has a good summary.

Here are some you may not have seen.

29.1 Arithmetic Operators

29.1.1 floor division //

```
5 // 2
```

```
2
```

```
-5 // 2
```

```
-3
```

```
5.5 // 2
```

```
2.0
```

29.1.2 modulus %

Returns the remainder

```
5 % 2
```

```
1
```

odd integers % 2 = 1
even integers % 2 = 0

Look at this …

```
5.5 / 2, 5.5 // 2, 5.5 % 2
```

```
(2.75, 2.0, 1.5)
```

29.1.3 exponentiation **

```
5**3
```

```
125
```

29.2 String Operators

29.2.1 concatenation +

The plus sign is an **ovderloaded** operator in Python.

```
myString = 'This: '
```

```
my2ndString = myString + ' Goodbye, world!'
```

```
my2ndString
```

```
'This:  Goodbye, world!'
```

29.2.2 repetition *

```
# print('-' * 80)
```

```
myString*2
```

```
'This: This: '
```

```
myString * 5
```

```
'This: This: This: This: This: '
```

```
bart_S1E3 = 'I will not skateboard in the halls'
```

```
print((bart_S1E3 + '\n') * 5)
```

```
I will not skateboard in the halls
I will not skateboard in the halls
I will not skateboard in the halls
I will not skateboard in the halls
I will not skateboard in the halls
```

```
print('-' * 80)
```

```
--------------------------------------------------------------------------------
```

See them all :-)

29.3 Assignment Operator =

We've used this already, but it too is an operator.

```
epoch = 20
print('epoch:', epoch)
```

```
epoch: 20
```

29.4 Comparison Operators

Comparisons are questions.

They return a boolean value.

29.4.1 equality ==

```
0 == (10 % 5)
```

```
True
```

```
'Boo' == 'Hoo'
```

```
False
```

Can we compare strings

```
'A' < 'B'
```

```
True
```

```
ord('A'), ord('B')
```

```
(65, 66)
```

29.4.2 inequality !=

```
5/9 != 0.5555
```

```
True
```

29.5 Logical Operators

Python uses words where other languages will use other symbols.

29.5.1 Conjunctions `and`, `or`, `not`

Note the we group comparisons with parentheses.

```
x = 10

(x % 10 == 0) or (x < -1)
```

```
True
```

```
(x % 10 == 0) and (x < -1)
```

```
False
```

```
not x == 5
```

```
True
```

29.5.2 Identity `is`

The `is` keyword is used to test if two variables refer to the same object.

The test returns `True` if the two objects are the same object.

The test returns False if they are not the same object, even if the two objects are 100% equal.

Use the `==` operator to test if two variables are equal.

– from W3Schools on Identity Operators

`is`

```
x = 'fail'
```

```
x is 'fail'
```

```
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
/var/folders/14/rnyfspnx2q131jp_752t9fc80000gn/T/ipykernel_53814/1139635342.py:1: SyntaxWarn
  x is 'fail'
```

```
True
```

is not

```
x is not 'fail'
```

```
<>:1: SyntaxWarning: "is not" with a literal. Did you mean "!="?
<>:1: SyntaxWarning: "is not" with a literal. Did you mean "!="?
/var/folders/14/rnyfspnx2q131jp_752t9fc80000gn/T/ipykernel_53814/1754352910.py:1: SyntaxWarn
  x is not 'fail'
```

```
False
```

```
x = 'foo'
y = 'foo'
x is y
```

```
True
```

```
x = ['a']
y = ['a']
x is y
```

```
False
```

29.5.3 Negation `not`

```
not True, not False, not 0, not 1, not 1000, not None
```

```
(False, True, True, False, False, True)
```

30 Unary Operators

Python offers a short-cut for most operators. When updating a variable with an operation to that variable, such as:

```
my_var = my_var + 1  # Incrementing
```

You can do this:

```
my_var += 1
```

Python supports many operators this way. Here are some:

```
a -= a
a \= a
a \\= a
a %= a
a *= a
a **= a
```

31 Expressions

Variables, literal values, and operators are the building blocks of ebxpressions.

For example, the following combines three operators and four variables:

```
1 + 2 * 3 / 2
```

```
4.0
```

Python employs **operator precedence** when evaluating expressions:

```
P - Parentheses
E - Exponentiation
M - Multiplication
D - Division
A - Addition
S - Subtraction
```

You can use parentheses to group them to force the order of operations you want:

```
(1 + 2) * (3 / 2)
```

```
4.5
```

Variables and literal values can be combined:

```
y = 5
m = 2.5
b = 10
```

```
y = m * 10 + b
y
```

```
35.0
```

```
y = m * 5 + b
y
```

```
22.5
```

Expresssion can be very complex.

Expressions evaluate to a value, just as single variables do.

Therefore, they can be put anywhere a value is accepted.

```
int((y + 10) ** 8)
```

```
1244706300354
```

32 NB: Numbers

These are built-in mathematical functions for numbers.

32.1 `pow()` Power

```
pow(2,3) # 2 raised to 3 = 8
```

32.2 `abs()` Absolute value

```
abs(-2) # returns 2, the absolute value of its argument
```

32.3 `round()` Round

Rounding up or down its argument (to closest whole number).

```
round(2.8) # rounds up to 3.0
```

```
3
```

```
round(1.1) # rounds down to 1.0
```

```
1
```

33 Math library functions

See the Python docs on the math library.

```
import math
```

33.1 `math.sqrt()` Square root

```
math.sqrt?
```

```
Signature: math.sqrt(x, /)
Docstring: Return the square root of x.
Type:      builtin_function_or_method
```

```
# sqrt(intOne)
```

```
math.sqrt(12) # using the square-root function from the math library
```

```
3.4641016151377544
```

```
print(math.floor(2.5)) # returns largest whole number less than the argument
print(math.floor(2.9))
print(math.floor(2.1))
```

```
2
2
2
```

33.2 `math.log()`

```python
math.log?
```

```python
math.log(100, 10)
```

```python
math.log(256, 2)
```

34 The Random library

See random — Generate pseudo-random numbers for more info.

```
import random
```

34.1 `random.random()`

```
random.random?
```

```
print(random.random()) # using random() function in random library
    # will return a number between 0 and 1
```

34.2 `random.randint()`

```
random.randint?
```

```
print(random.randint(1,100)) # specify a range in the parenthesis
    # this will return a random integer in the range 1-100
```

35 NB: Booleans

A `boolean` value takes one of `True` or `False`, which are built-in values

check if `cache` is True, using `if` statement
`if` statement using a bool evaluates to True or False

```
cache = True

if cache:
   print('data will be cached')
```

```
data will be cached
```

```
print(type(cache))
```

```
<class 'bool'>
```

Booleans are frequently used in `if/then` statements.

We'll cover these later.

```
cache = True
oome = False

if cache or oome:
    print('condition met!')
else:
    print("No dice.")
```

AND statements will short circuit if an early condition fails.

```
if oome and cache:
    print('condition met!')
```

In this case, since *oome* is False, the check on *cache* never happens.

36 NB: Strings

Strings are signified by quotes.

Single and double quotes are identical in function.

They must be “straight quotes” though – cutting and pasting from a Word document with smart quotes won’t work.

```
'hello world!' == "hello world!"
```

36.1 Quote prefixes

36.1.1 `r` strings

Prefixing a string causes escape characters to be uninterpreted.

```
print("Sentence one.\nSentence two.")
```

```
print(r"Sentence one.\nSentence two.")
```

36.1.2 `f` strings

Prefixing a string with `f` allows variable interpolation – inplace evaluation of variables in strings.

```
ppl = 'knights'
greeting = 'Ni'
```

```
print(f'We are the {ppl} who say {greeting}!') # Output: We are the knights who say Ni!
```

The brackets and characters within them (called format fields) are replaced with the passed objects.

```
print(b"This is a sentence.")
```

```
print("This is a sentence.")
```

37 Printing `print()`

Python uses a print function.

```
print("This is a simple print statement")
```

Python supports special "escape characters" in strings that produce effects when printed.

```
\\     Backslash (\)
\'     Single quote (')
\"     Double quote (")
\n     ASCII Linefeed, aka new line
```

Note that these are not unique to Python. They are part of almost all languages.

```
# Tab character ( \t )
print("Hello,\tWorld! (With a tab character)")
```

```
Hello,  World! (With a tab character)
```

```
# Inserting a new line (line feed) character ( \n )
print("Line one\nLine two, with newline character")
```

```
Line one
Line two, with newline character
```

```
# Concatenation in strings:
# Use plus sign ( + ) to concatenate the parts of the string
print("Concatenation," + "\t" + "in strings with tab in middle")
```

```
# If you wanted to print special characters
# Printing quotes
print('Printing "quotes" within a string') # mixing single and double quotes
```

```python
# What if you needed to print special characters like (\) or (') or (")
print('If I want to print \'single quotes\' in a string, use backslash!')
print("If I want to print \"double quotes\" in a string, use backslash!")
print('If I want to print \\the backslash\\ in a string, also use backslash!')
```

```
If I want to print 'single quotes' in a string, use backslash!
If I want to print "double quotes" in a string, use backslash!
If I want to print \the backslash\ in a string, also use backslash!
```

The print function puts spaces between strings and a newline at the end, but you can change that:

```python
print("This", "is", "a", "sentence")
```

```python
print("This", "is", "a", "sentence", sep="--")
```

```python
print("This", "is", "a", "sentence")
print("This", "is", "a", "sentence")
```

```python
print("This", "is", "a", "sentence", end=" | ")
print("This", "is", "a", "sentence")
```

38 Comments

Comments are lines of code that aren't read by the interpreter.

They are used to explain blocks of code, or to remove code from execution when debugging.

```
# This is single-line comment
```

These following are multiline strings that can serve as comments:

```
foo = '''
This is an
example of
a multi-line
comment: single quotes
'''
```

```
foo
```

```
'\nThis is an\nexample of\na multi-line\ncomment: single quotes\n'
```

```
print(foo)
```

```

This is an
example of
a multi-line
comment: single quotes
```

```
"""
Here is another
example of
a multi-line
comment: double quotes
```

```
"""
```

Note that multiline comments also evaluate as values.

39 Run-time User Input

```
answer = input("What is your name? ")
print("Hello, " + answer + "!")
```

40 Some String Functions

```
Built-in string methods and functions.

See [Common String Operations](https://docs.python.org/3/library/string.html) for more inf
```

40.1 `.lower()`, `.upper()`

```
'BOB'.lower() #.upper()
```

```
'bob'
```

40.2 `.split()`

Parase a string based on a delimiter, which defaults to whitespace.

NOTE: This does *not* use regular expressions.

This returns a list.

```
montyPythonQuote = 'are.you.suggesting.coconuts.migrate'
```

```
'are.you.suggesting.coconuts.migrate'.split('.')
```

```
['are', 'you', 'suggesting', 'coconuts', 'migrate']
```

```
montyPythonQuote
```

```
'are.you.suggesting.coconuts.migrate'
```

```
montyPythonQuote.split('.') # split by the '.' delimiter. Result: a list!
```

```
['are', 'you', 'suggesting', 'coconuts', 'migrate']
```

40.3 `.strip()`, `.rstrip()`, `lstrip()` Strip methods

Strip out extra whitespace using strip(), rstrip() and lstrip() functions

`.strip()` removes white space from anywhere
`.rstrip()` only removes white space from the right-hand-side of the string
`.lstrip()` only removes white space from the left-hand-side of the string

```
str1 = '  hello, world!'    # white space at the beginning
str2 = '  hello, world!  '  # white space at both ends
str3 = 'hello, world!  '    # white space at the end
```

```
str1, str2, str3
```

```
str1.lstrip(), str1.rstrip()
```

```
str2.strip(), str2.rstrip()
```

```
str2.lstrip(), str3.rstrip()
```

```
status.startswith('a')
```

```
status.endswith('s')
```

40.4 `.replace()`

```
"latina".replace("a", "x")
```

40.5 `.format()`

Variable values can be embedding in strings using the `format()` function.
Place {} in the string in order from left to right. followed by `.format(var1, var2, ...)`‘

```
epoch = 20
loss = 1.55

print('Epoch: {}, loss: {}'.format(epoch, loss))
```

This breaks, as three variables are required based on number of {}

```
print('Epoch: {}, loop: {}, loss: {}'.format(epoch, loss))
```

40.6 `.zfill()`

Basic usage of the str.zfill() method (pads a numeric string on the left with zeros) It understands about plus and minus signs

```
print('12'.zfill(5))       # Output: 00012
print('-3.14'.zfill(7))    # Output: -003.14
print('3.141592'.zfill(5)) # Output: 3.141592
```

41 Strings are Lists

Actually, they are list-like.

Here are some functions applicable to strings because they are lists.

41.1 `len()` Length

This is built-in length funciton tells us how many characters in the string.

It also applys to any list-like object, including strings, lists, dicts, sets, and dataframes.

```
len?
```

```
my_new_tring = 'This is a string'
```

```
len(my_new_tring)
```

41.1.1 Indexing

Since strings are sequences in Python, each character of the string has a unique position that can be indexed.

Indexes are indicated by suffixed brackets, e.g. `foo[]`

```
my_new_tring[0] # displays the first character of the string
            # first position is position zero. Will display 'h'
```

```
my_new_tring[-1] # displays the last character. Negatives count backwords.
```

41.1.2 Slicing

We can used the colon to 'slice' strings (and lists)

```python
my_new_tring[0:4] # First four characters (index positions 0-3)
```

```python
my_new_tring[:4]  # Beginning (0) to (n-1) position
```

```python
my_new_tring[4:]  # Fifth character and onwards until the end of the string
```

it is NOT possible to reassign elements of a string. Python strings are **immutable**.

```python
status = 'success'
status[0] = 't'
```

Add strings and handle pathing

42 NB: Structures

In contrast to primitive data types, data structures organize types into structures that have certain properties, such as **order**, **mutability**, and **addressing scheme**, e.g. by index.

A list is an ordered sequence of items.

Each element of a list is associated with an integer that represents the order in which the element appears.

Lists are indexed with **brackets** `[]`.

List elements are accessed by providing their order number in the brackets.

Lists are **mutable**, meaning you can modify them after they have been created.

They can contain mixed types.

42.1 Constructing

They can be **constructed** in several ways:

```
list1 = []
list2 = list()
list3 = "some string".split()
numbers = [1,2,3,4]
```

42.2 Indexing

Zero-based indexing

Python uses xzero-based indexing, which means for a collection `mylist`

`mylist[0]` references the first element
`mylist[1]` references the second element, etc

For any iterable object of length *N*:
`mylist[:n]` will return the first *n* elements from index *0* to *n-1*
`mylist[-n:]` will return the last *n* elements from index *N-n* to *N-1*

```
numbers[0] # Access first element (output: 1)
```

```
1
```

```
numbers[-1]
```

```
4
```

```
numbers[0] + numbers[3] # doing arithmetic with the values (output: 5)
```

```
5
```

```
numbers[len(numbers)]
```

```
IndexError: list index out of range
```

42.3 Slicing

```
numbers[0:2] # Output: [1, 2]
```

```
[1, 2]
```

```
numbers[1:3] # Output: [2, 3]
```

```
[2, 3]
```

```
len(numbers) # use len() function to find the size. Output: 4
```

```
4
```

```
numbers[2:]  # Output: [3, 4]
```

```
[3, 4]
```

42.4 Multiply lists by a scalar

A scalar is a single value number.

```
numbers * 2
```

42.5 Concatenate lists with +

```
numbers2 = [30, 40, 50]
```

```
numbers + numbers2 # concatenate two lists
```

42.6 Lists can mix types

```
myList = ['coconuts', 777, 7.25, 'Sir Robin', 80.0, True]
```

```
myList
```

What happens if we multiply a list with strings?

```
# myList * 2
```

42.7 Lists can be nested

```
names = ['Darrell', 'Clayton', ['Billie', 'Arthur'], 'Samantha']
names[2] # returns a *list*
names[0] # returns a *string*
```

cannot subset into a float, will break

```
names[2][0]
```

42.8 Lists can concatenated with +

```
variables = ['x1', 'x2', 'x3']
response = ['y']
```

```
variables + response
```

```
['x1', 'x2', 'x3', 'y']
```

43 Dictionaries `dict`

Like a hash table.

Has key-value pairs.

Elements are indexed using brackets `[]` (like lists).

But they are constructed used braces `{}`.

Key names are unique. If you re-use a key, you overwrite its value.

Keys don't have to be strings – they can be numbers or tuples or expressions that evaluate to one of these.

43.1 Constructing

```
dict1 = {
    'a': 1,
    'b': 2,
    'c': 3
}
```

```
dict2 = dict(x=55, y=29, z=99) # Note the absence of quotes around keys
```

```
dict2
```

```
{'x': 55, 'y': 29, 'z': 99}
```

```
dict3 = {'A': 'foo', 99: 'bar', (1,2): 'baz'}
```

```
dict3
```

```
{'A': 'foo', 99: 'bar', (1, 2): 'baz'}
```

43.2 Retrieve a value

Just write a key as the *index*.

```
phonelist = {'Tom':123, 'Bob':456, 'Sam':897}
```

```
phonelist['Bob']
```

43.3 Print list of keys, values, or both

Use the `.keys()`, `.values()', or`.items()‘ methods.

Keys are not sorted. For example, they are not ordered in order in which they were added.

```
phonelist.keys() # Returns a list
```

```
phonelist.values() # Returns a list
```

```
phonelist.items() # Returns a list of tuples
```

```
phonelist # note the data returned is not the same as the data entered
```

44 Tuples

A tuple is like a list but with one big difference: **a tuple is an immutable object!**

You can't change a tuple once it's created.

A tuple can contain any number of elements of any datatype.

Accessed with brackets `[]` but constructed with parentheses `()`.

```
numbers
```

44.1 Constructing

Created with comma-separated values, with or without parenthesis.

```
letters = 'a', 'b', 'c', 'd'
```

```
letters
```

```
numbers = (1,2,3,4) # numbers 1,2,3,4 stored in a tuple
```

A single valued tuple must include a comma `,`, e.g.

```
tuple0 = (29)
```

```
tuple0, type(tuple0)
```

```
tuple1 = (29,)
```

```
tuple1, type(tuple1)
```

```
len(numbers)
```

```
numbers[0] = 5 # Trying to assign a new value 5 to the first position
```

45 Common functions and methods to all sequences

```
len()
in
+
*
```

```
[1, 3] * 8
```

```
(1, 3) * 8
```

45.1 Membership with `in`

Returns a boolean.

```
'Sam' in phonelist
```

46 Sets

A `set` is an unordered collection of unique objects.

They are subject to set operations.

```
peanuts = {'snoopy','snoopy','woodstock'}
```

```
peanuts
```

Note the set is deduped

Since sets are unordered, they don't have an index. This will break:

```
peanuts[0]
```

```
for peanut in peanuts:
    print(peanut)
```

Check if a value is in the set using `in`

```
'snoopy' in peanuts
```

Combine two sets

```
set1 = {'python','R'}
set2 = {'R','SQL'}
```

This fails:

```
set1 + set2
```

This succeeds:

```
set1.union(set2)
```

Get the set intersection

```
set1.intersection(set2)
```

47 Ranges

A range is a sequence of integers, from `start` to `stop` by `step`. - The `start` point is zero by default.
- The `step` is one by default.
- The `stop` point is NOT included.

Ranges can be assigned to a variable.

```python
rng = range(5)
```

More often, ranges are used in iterations, which we will cover later.

```python
for rn in rng:
    print(rn)
```

another range:

```python
rangy = range(1, 11, 2)
for rn in rangy:
    print(rn)
```

48 Collections and `defaultdict`

Very often you will want to build a dictionary from some data source, and add keys as they appear. The default `dict` type in Python, however, requires that the key exists before you can mutate it. The `defaultdict` type in the `collections` module solves this problem. Here's an example.

```
source_data = """
Lorem Ipsum is simply dummy text of the printing and typesetting industry.
Lorem Ipsum has been the industry's standard dummy text ever since the 1500s,
when an unknown printer took a galley of type and scrambled it to make a type
specimen book. It has survived not only five centuries, but also the leap
into electronic typesetting, remaining essentially unchanged. It was
popularised in the 1960s with the release of Letraset sheets containing
Lorem Ipsum passages, and more recently with desktop publishing software
like Aldus PageMaker including versions of Lorem Ipsum.
"""[1:-1].split()
```

```
# source_data
```

48.1 Try with `dict`

```
words = {}
for word in source_data:
    words[word] += 1
```

48.2 Use `try` and `except`

```
for word in source_data:
    try:
        words[word] += 1
    except KeyError:
        words[word] = 1
```

```
words
```

48.3 Or use `.get()`

```
for word in source_data:
    words[word] = words.get(word, 0) + 1
```

48.4 Use `collections.defaultdict`

```
from collections import defaultdict
```

```
words2 = defaultdict(int) # Not the type must be set
```

```
for word in source_data:
    words2[word] += 1
```

```
words2
```

Part IV

M03 Control Structures

Topics

- More on Statements and Syntax
- Control Structures and Loops
- Iterators
- Comprehensions

Outcomes

- Recognize primary control structures available in Python and their basic use cases
- Write comprehensions for each of Python's list-like data structures
- Recognize when iterators are used by Python functions (such as open())
- Understand basic conditional logic statements and their role in designing data flow in a program

Readings

Required

Lutz, 2019, Part III, Chapter 10. Introducing Python Statements

Lutz, 2019, Part III, Chapter 11. Assignments, Expressions, and Prints Read only up to and including "The Python 3.X print Function."

Lutz, 2019, Part III, Chapter 12. if Tests and Syntax Rules

Lutz, 2019, Part III, Chapter 13. while and for Loops

Lutz, 2019, Part III, Chapter 14. Iterations and Comprehensions

Lutz, 2019, Part III, Chapter 15: The Documentations Interlude

Optional

Variables, Expressions, Statements, Types (Python Notes)

More Control Flow Tools (Python Docs)

If … Then (W3S)

Iterators (GFG)

49 Values, Variables, Expressions, and Statements

49.1 Definitions

- **Values**: Raw data elements represented in a program, e.g. numbers and strings.
- **Variables**: Names to which values are assigned.
- **Expressions**: Combinations of values, variables, operators, functions, and other expressions that evaluate to a value.
- **Statements**: Groupings of expressions that produce some result. Statements **do** things.

49.2 Statement Types

See Statement Types in Lutz.

50 NB: Control Structures

Topics:

- conditional statements
- if, else, elif
- for-loop
- while-loop
- break
- continue
- iteration

50.1 Introducing Control Structures

Python includes structures to control the flow of a program:

- `conditions` (if, else)
- `loops`
  - `while-loop`
    Execute statements while a condition is true
  - `for-loop`
    Iterates over a iterable object (list, tuple, dict, set, string)

50.2 Indentation

This is where Python differs from most languages. To define control structures,
and functional blocks of code in general, most languages use either characters like braces `{` and `}` or key words like `IF ... END IF`.

Python uses tabs – spaces, actually – to signify logical blocks off code.

It is therefore imperative to understand and get a feel for indentation. For more information, see Lutz 2019, "A Tale of Two Ifs."

50.3 Conditions

50.3.1 `if` and `else` can be used for conditional processing.

```
val = -2

if val >= 0:
    print(val)
else:
    print(-val)
```

```
2
```

50.3.2 `elif`

`elif` is reached when the previous statements are not.

```
val = -2

if -10 < val < -5:
    print('bucket 1')
elif -5 <= val < -2:
    print('bucket 2')
elif val == -2:
    print('bucket 3')
```

```
bucket 3
```

50.3.3 `else`

`else` can be used as a catchall

```
val = 5

if -10 < val < -5:
    print('bucket 1')
elif -5 <= val < -2:
    print('bucket 2')
elif val == -2:
    print('bucket 3')
else:
    print('bucket 4')
```

```
bucket 4
```

50.3.4 `if` and `else` as one-liners

```
x = 3
print('odd') if x % 2 == 1 else print('even')
```

```
odd
```

Notice `==` for checking the condition x % 2 == 1.

both `if` and `else` are required. This breaks:

```
print('odd') if x % 2 == 1
```

```
SyntaxError: invalid syntax (471325368.py, line 1)
```

50.3.5 Using multiple conditions

If statements can be complex combinations of expressions.

Use parentheses carefully, to keep order of operations correct.

```
## correct

val = 2

if (-2 < val < 2) or (val > 10):
    print('bucket 1')
else:
    print('bucket 2')
```

```
bucket 2
```

```
## incorrect - misplaced parenthesis

if (-2 < val) < 2 or val > 10:
    print('bucket 1')
else:
    print('bucket 2')
```

```
bucket 1
```

and this is because True $<$ 2, as True is cast to integer value 1

this is not the desired result…but does it make sense?

50.4 Loops

50.4.1 `while`

What does this print?

```
ix = 1
while ix < 10:
    ix = ix * 2
print(ix)
```

```
16
```

50.4.2 `break` to exit the loop altogether

sometimes you want to quit the loop early, if some condition is met.
uses `if-statement`

```
ix = 1
while ix < 10:
    ix = ix * 2
    if ix == 4:
        break
print(ix)
```

```
4
```

The `break` causes the loop to end early

50.4.3 `continue` to stop the current iteration

sometimes you want to introduce skipping behavior in the loop.
uses `if-statement`

```
ix = 1
while ix < 10:
    ix = ix * 2
    if ix == 4:
        print('skipping 4...')
        continue
    print(ix)
```

```
2
skipping 4...
8
16
```

The `continue` causes the loop to skip printing 4

50.4.4 `for`

iterate over an iterable

```
cities = ['Charlottesville','New York','SF','BOS','LA']

for city in cities:
    city = city.lower()
    print(city)
```

```
charlottesville
new york
sf
bos
la
```

quit early if `SF` reached, using **break**

```
cities = ['Charlottesville','New York','SF','BOS','LA']

for city in cities:
    if city == 'SF':
        break
    city = city.lower()
    print(city)
```

```
charlottesville
new york
```

skip over `SF` if reached, using **continue**

```
cities = ['Charlottesville','New York','SF','BOS','LA']

for city in cities:
    if city == 'SF':
        continue
    city = city.lower()
    print(city)
```

```
charlottesville
new york
bos
la
```

50.5 `while` vs `for`

For loops are used to loop through a list of values or an operation in which the number of iterations is **known** in advance.

While loops are when **you don't know** how many interations it will take – you are depending on some condition to be met.

It is possible for while loops to be unending, for example:

```
while 1:
    print("This is so annoying")
```

51 NB: Iterables and Iterators

Purpose - Define iterables and iterators - Using two methods, show how iterators can be used to return data from sets, lists, strings, tuples, dicts: - `for` loops
- `iter()` and `next()`

Specific Topics - iterable objects or iterables - iterators - iteration - sequence - collection

52 Defining Iterables and Iterators

Iterable objects or **iterables** can return elements one at a time.

An **iterator** is an object that iterates over iterable objects such as sets, lists, tuples, dictionaries, and strings.

Iteration can be implemented: - with a `for` loops - with the `next()` method

Next, we show examples for various iterables.

53 Lists

53.1 iterating using `for`

```
tokens = ['living room', 'was', 'quite', 'large']

for tok in tokens:
    print(tok)
```

```
living room
was
quite
large
```

53.2 iterating using `iter()` and `next()`

`iter()` gets an iterator. Pops out a value each time it's used.

`next()` gets the next item from the iterator

```
tokens = ['living room','was','quite','large']
myit = iter(tokens)
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

```
living room
was
quite
large
```

Calling `next()` when the iterator has reached the end of the list produces an exception:

```python
print(next(myit))
```

```
StopIteration:
```

Next, look at the type of the iterator, and the documentation

```python
type(myit)
```

```
list_iterator
```

```python
# help(myit)
```

```python
help(next)
```

```
Help on built-in function next in module builtins:

next(...)
    next(iterator[, default])

    Return the next item from the iterator. If default is given and the iterator
    is exhausted, it is returned instead of raising StopIteration.
```

Note that `for` implicitly creates an iterator and executes `next()` on each loop iteration. This is best way to iterate through a list-like object.

54 Sequences and Collections

We iterated over a list. Next we will illustrate for other iterables: `str`, `tuple`, `set`, `dict`

lists, tuples, and strings are **sequences**. Sequences are designed so that elements come out of them in the same order they were put in.

Sets and dictionaries are not sequences, since they don't keep elements in order. They are called **collections**. The ordering of the items is arbitrary.

NOTE: This has changed for dictionaries in Python 3.7: > the insertion-order preservation nature of dict objects has been declared to be an official part of the Python language spec.
– What's New in Python 3.7

55 Sets

iterating using `for`

```python
princesses = {'belle','cinderella','rapunzel'}

for princess in princesses:
    print(princess)
```

```
cinderella
belle
rapunzel
```

iterating using `iter()` and `next()`

```python
princesses = {'belle','cinderella','rapunzel'}

myset = iter(princesses) # note: set has no notion of order
print(next(myset))
print(next(myset))
print(next(myset))
```

```
cinderella
belle
rapunzel
```

56 Strings

iterating using `for`

```python
strn = 'data'

for s in strn:
    print(s)
```

```
d
a
t
a
```

iterating using `iter()` and `next()`

```python
st = iter(strn)

print(next(st))
print(next(st))
print(next(st))
print(next(st))
```

```
d
a
t
a
```

57 Tuples

iterating using `for`

```
metrics = ('auc','recall','precision','support')

for met in metrics:
    print(met)
```

```
auc
recall
precision
support
```

iterating using `iter()` and `next()`

```
metrics = ('auc','recall','precision','support')

tup_metrics = iter(metrics)
print(next(tup_metrics))
print(next(tup_metrics))
print(next(tup_metrics))
print(next(tup_metrics))
```

```
auc
recall
precision
support
```

58 Dictionaries

iterating using `for`

```python
courses = {'fall':['regression','python'], 'spring':['capstone','pyspark','nlp']}
```

```python
# iterate over keys
for k in courses:
    print(k)
```

```
fall
spring
```

```python
# iterate over keys, using keys() method
for k in courses.keys():
    print(k)
```

```
fall
spring
```

```python
# iterate over values
for v in courses.values():
    print(v)
```

```
['regression', 'python']
['capstone', 'pyspark', 'nlp']
```

```python
# iterate over keys and values using `items()`
for k, v in courses.items():
    print(f"{k}:\t{', '.join(v)}")
```

```
fall:   regression, python
spring: capstone, pyspark, nlp
```

Alternatively, keys and values can be extracted from the dict by: - looping over the keys - extract the value by indexing into the dict with the key

```
# iterate over keys and values using `key()`.
for k in courses.keys():
    print(f"{k}:\t{', '.join(courses[k])}") # index into the dict with the key
```

```
fall:   regression, python
spring: capstone, pyspark, nlp
```

59 Ranges

iterating using `for`

If you just want to iterate for a known number of times, use `range()`.

```
for i in range(10):
    print(str(i+1).zfill(2), (i+1)**2 * '|')
```

```
01 |
02 ||||
03 |||||||||
04 ||||||||||||||||
05 |||||||||||||||||||||||||
06 ||||||||||||||||||||||||||||||||||||
07 |||||||||||||||||||||||||||||||||||||||||||||||||
08 ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
09 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
10 |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

60 Get iteration number with `enumerate()`

Very often you will want to know iteration number you are on in a loop.

This can be used to name files or dict keys, for example.

`enumerate()` will return the index and key for each iteration.

```
courses
```

```
{'fall': ['regression', 'python'], 'spring': ['capstone', 'pyspark', 'nlp']}
```

```
for i, semester in enumerate(courses):
    course_name = f"{str(i).zfill(2)}_{semester}:\t{'-'.join(courses[semester])}"
    print(course_name)
```

```
00_fall:    regression-python
01_spring:  capstone-pyspark-nlp
```

61 Nested Loops

Iterations can be nested!

This works well with nested data structures, like dicts within dicts.

This is basically how `JSON` files are handled, BTW.

Be careful, though – these can get deep and complicated.

```
for i, semester in enumerate(courses):
    print(f"{i+1}. {semester.upper()}:")
    for j, course in enumerate(courses[semester]):
        print(f"\t{i+1}.{j+1}. {course}")
```

```
1. FALL:
    1.1. regression
    1.2. python
2. SPRING:
    2.1. capstone
    2.2. pyspark
    2.3. nlp
```

iterating using `iter()` and `next()`

62 NB: Comprehensions

Purpose - Explain the benefit of list comprehensions - Illustrate the use of list comprehensions - Explain the benefit of dict comprehensions - Illustrate the use of dict comprehensions

Concepts - list comprehension - dict comprehension - iterators

63 List Comprehensions

Consider this task: check if each integer in a list is odd.

Without list comprehensions, you might do this:

63.1 Check if Odd

```
vals = [1,5,6,8,12,15]
is_odd = []

for val in vals:
    if val % 2: # if remainder is one, val is odd
        is_odd.append(True)
    else:       # else it's not odd
        is_odd.append(False)

is_odd
```

```
[True, True, False, False, False, True]
```

The code loops over each value in the list, checks the condition, and appends to a new list.

The code works, but it's lengthy compared to a list comprehension.

The approach takes extra time to write and understand.

Let's solve with a list comprehension:

```
is_odd = [val % 2 == 1 for val in vals]
is_odd
```

```
[True, True, False, False, False, True]
```

Much shorter, and if you understand the syntax, quicker to interpet.

Note the in-place use of an expression.

Now let's discuss the syntax.

64 Comprehensions in General

Comprehensions provide a concise method for iterating over any list-like object to a new list like object.

There are comprehensions for each list-like object: * List comprehensions * Dictionary comprehensions * Tuple comprehensions * Set comprehensions

Comprehensions are essentially very concise `for` loops. They are compact visually, but they also are more efficient than loops.

All comprehensions have the form:

listlike_result = `[ expression + context]`

The type of comprehension is indicated by the use of enclosing pairs, just like anonymous constructors:

- List comprehensions `[expression + context]`
- Dictionary comprehensions `{expression + context}`
- Tuple comprehensions `(expression + context)`
- Set comprehensions `{expression + context}`

Expression defines what to do with each element in the list. This has the structure of the kind of comprehension. So, dictionary comprehension expressions take the form `k:v` while sets use `v`.

Context defines which list elements to select. The context always consists of an arbitrary number of `for` and `if` statements.

65 More examples

65.1 Stop Word Remover

Create list of words, and list of stop words.
Filter out the stop words (considered not important).

```
stop_words = ['a','am','an','i','the','of']
words      = ['i','am','not','a','fan','of','the','film']

clean_words = [wd for wd in words if wd not in stop_words]
clean_words
```

```
['not', 'fan', 'film']
```

placing the color-coding on the list comprehension:

[wd   for wd in words if wd not in stop_words]

- the expression is very simple: **wd**. keep the word if meets condition
- the condition does the work: if the word isn't in list of stop words, keep it

Side note: This task can also be done with sets, if you are not concerned with mulitple instances of the same word:

```
s1 = set(stop_words)
s2 = set(words)
s3 = s2 - s1
```

```
s3
```

```
{'fan', 'film', 'not'}
```

65.2 Select Tokens Containing Units

Given a list of measurements, retain elements containing mmHg (millimeters of mercury)

```
units = 'mmHg'
measures = ['20', '115mmHg', '5mg', '10 mg', '7.5dl', '120 mmHg']
meas_mmhg = [meas for meas in measures if units in meas]
```

```
meas_mmhg
```

```
['115mmHg', '120 mmHg']
```

Filtering on two conditions

```
units1 = 'mmHg'
units2 = 'dl'
meas_mmhg_dl = [meas for meas in measures if units1 in meas or units2 in meas]
```

```
meas_mmhg_dl
```

```
['115mmHg', '7.5dl', '120 mmHg']
```

This can be written differently for clarity:

```
[meas
 for meas in measures
 if units1 in meas
 or units2 in meas]
```

```
['115mmHg', '7.5dl', '120 mmHg']
```

66 Dictionary Comprehensions

Dictionary comprehensions provide a concise method for iterating over a dictionary to create a new dictionary.

This is common when data is structured as key-value pairs, and we'd like to filter the dict.

```
# various deep learning models and their depths

model_arch = {'cnn_1':'15 layers', 'cnn_2':'20 layers', 'rnn': '10 layers'}
```

```
# create a new dict containing only key-value pairs where the key contains 'cnn'

cnns = {key:model_arch[key] for key in model_arch.keys() if 'cnn' in key}
cnns
```

```
{'cnn_1': '15 layers', 'cnn_2': '20 layers'}
```

We build the key-value pairs using `key:model_arch[key]`, where the key indexes into the dict `model_arch`

Part V

M04 Functions

Topics

- Built-in functions
- User-defined functions
- Variable scope
- Lambda functions
- Design of functions
- Recursion

Outcomes

- Be able to use Pythons native and imported functions
- Be able to write your own functions
- Understand concept of variable scope
- Be able to write lambda functions and understand their use cases
- Grasp basic principles of function design
- Implement simple recursion functions

Readings

Required

Lutz 2019, Part IV, Chapter 16: Function Basics

Lutz 2019, Part IV, Chapter 17: Scopes Non-local is for advanced users

Lutz 2019, Part IV, Chapter 18. Arguments

Lutz 2019, Part IV, Chapter 19: Advanced Function Topics

Optional

McKinney, Python for Data Analysis, Appendix A: Python Language Essentials

Read section on Functions

Functions (W3S)

Global and Local Variables (GFG)

Lambda Functions (Real Python)

67 NB: Introduction to Functions

Objectives - Explain the benefits of functions - Illustrate how to use built-in functions - Illustrate how to create and use your own (user-defined) functions - Demonstrate the scope and lifetime of a variable - Illustrate global and local nature of variables through functions - Demonstrate function parameter use - Provide recommendations on how to create and document functions - Show how to print and write docstrings

Concepts - functions - built-in functions - user-defined functions - variable scope - global versus local variables - default arguments - *args - function call - docstring

67.1 Introduction

A function is piece of source code, separate fom the larger program, that performs a specific task.

This section of code is given a name and can be called from the main program. It is called by using its given name.

Functions are the **verbs** of a programming language. They signify action, and take subjects and objects (as it were).

Functions take **input** and produce **output**.

- Function inputs are called both **parameters** and **arguments**.
- Outputs are called **return** values

Functions are always written with parentheses at the end of their names, e.g.

`len(some_list)`

Internally, they contain a block of code to do their work.

Often the producte a **transformation** … from simple to complex.

When you use a function, we say you **call** a function. Programmers speak of “function calls” and “callbacks”.

67.2 Benefits

Reduce complex tasks into simpler tasks.

Eliminate duplicate code – no need to re-write, reuse function as needed.

Code reuse. Once function is written, you can reuse it in any other program.

Distribute tasks to multiple programmers. For example, each function can be written by someone.

Hide implementation details, i.e. abstraction.

Increase code readability.

Improve debugging by improving traceability. Things are easier to follow; you can jump from function to function.

67.3 Built-in Functions

Python provides many **built-in** functions. See Python built-in functions.

We've looked at many of these already.

These are functions that are available to use any time your are running Python.

To take one simple example, this is a built-in function: `bool()`.

Takes an argument x and returns a boolean value, i.e. `True` or `False`.

```
bool(0), bool(500)
```

67.4 Imported Functions

Python is meant to be a highly modular language.
It is not designed to have a lot of special purpose functions built into it.
These keeps it light and highly customizable.

Many functions (and other stuff) can be imported into a program to add to the functions that you can call in a script.

There are also many **packages** to bring in additional functions.

Packages and Libraries

67.5 User-Defined Functions

Python makes it easy for you to write your own functions. These are called **user-defined** functions.

Let's write a function to compare the list against a threshold.

```
def vals_greater_than_or_equal_to_threshold(vals, thresh):
    '''
    This is the "docstring" of a function. It is optional but expected. It describes it's
    purpose and the nature of the input and return values, as well as a sense of what it d
    More elaborate information should appear in external documentation packages with the f

    PURPOSE: Given a list of values, compare each value against a threshold

    INPUTS
    vals    list of ints or floats
    thresh  int or float

    OUTPUT
    bools  list of booleans
    '''

    bools = [val >= thresh for val in vals]

    return bools
```

Let's break down the components

The function definition starts with `def`, followed by name, one or more arguments in parenthesis, and then a colon.

Next comes a **docstring** to provide information to users about how and why to use the function.

The function **body** follows.

:astly is a `return` statement

The **function call** allows for the function to be used.
It consists of function name and required arguments:

`vals_greater_than_or_equal_to_threshold(arg1, arg2)` where `arg1`, `arg2` are arbitrary names.

67.5.1 About the docstring

A **docstring** m occurs as first statement in module, function, class, or method definition

Internally, it is saved in `__doc__` attribute of the function object.

It needs to be indented.

It can be a single line or a multi-line string.

67.5.2 Let's test our function

The function body used a `list comprehension` for the compare:

`[val >= thresh for val in vals]`

```
## validate that it works for ints

x = [3, 4]
thr = 4

vals_greater_than_or_equal_to_threshold(x, thr)
```

```
[False, True]
```

```
## validate that it works for floats

x = [3.0, 4.2]
thr = 4.2

vals_greater_than_or_equal_to_threshold(x, thr)
```

```
## vals_greater_than_or_equal_to_threshold("foo", "bar")
```

This gives correct results and does exactly what we want.

67.5.3 Users can print the docstring

```
print(vals_greater_than_or_equal_to_threshold.__doc__)
```

print the help

```
help(vals_greater_than_or_equal_to_threshold)
```

```
?vals_greater_than_or_equal_to_threshold
```

Let's test our function

The function body used a `list comprehension` for the comparison:

`[val >= thresh for val in vals]`

```
## validate that it works for ints

x = [3, 4]
thr = 4

vals_greater_than_or_equal_to_threshold(x, thr)
```

```
## validate that it works for floats

x = [3.0, 4.2]
thr = 4.2

vals_greater_than_or_equal_to_threshold(x, thr)
```

This gives correct results and does exactly what we want.

Print the docstring

```
print(vals_greater_than_or_equal_to_threshold.__doc__)
```

Print the help

```
help(vals_greater_than_or_equal_to_threshold)
```

Use the `?` prefix …

```
?vals_greater_than_or_equal_to_threshold
```

67.6 Passing Parameters

Functions need to be called with correct number of parameters.

This function requires two params, but the function call includes only one param.

```
def fcn_bad_args(x, y):
    return x + y
```

```
fcn_bad_args(10)
```

```
TypeError: fcn_bad_args() missing 1 required positional argument: 'y'
```

67.6.1 Parameter Order

When calling a function, **parameter order matters**.

```
def fcn_swapped_args(x, y):
    out = 5 * x + y
    return out
```

```
x = 1
y = 2
```

```
fcn_swapped_args(x, y)
```

```
7
```

```
fcn_swapped_args(y, x)
```

```
11
```

Generally it's best to keep parameters in order.

You can swap the order by putting the parameter names in the function call.

```
fcn_swapped_args(y=y, x=x)
```

67.6.2 Weirdness Alert

Note that the same name can be used for the parameter names and the variables passed to them.

The names themselves have nothng to do with each other!

In other words, just because a function names an argument `foo`,
the variables passed to it don't have to name `foo` or anything like it.
They can even be named the same thing – it does not matter.

67.7 Unpacking List-likes with `*args`

The `*` prefix operator can be passed to avoid specifying the arguments individually.

```
def show_arg_expansion(*models):

    print("models          :", models)
    print("input arg type  :",  type(models))
    print("input arg length:", len(models))
    print("-----------------------------")

    for mod in models:
        print(mod)
```

We can pass a tuple of values to the function …

```
show_arg_expansion("logreg", "naive_bayes", "gbm")
```

```
models          : ('logreg', 'naive_bayes', 'gbm')
input arg type  : <class 'tuple'>
input arg length: 3
-----------------------------
logreg
naive_bayes
gbm
```

You can also pass a list to the function.

If you want the elements unpacked, put `*` before the list.

```
models = ["logreg", "naive_bayes", "gbm"]
show_arg_expansion(*models)
```

```
models          : ('logreg', 'naive_bayes', 'gbm')
input arg type  : <class 'tuple'>
input arg length: 3
-----------------------------
logreg
naive_bayes
gbm
```

This approach allows your function to accept an arbitrary number of arguments.

```
show_arg_expansion('a b c d e f g'.split())
```

The reverse is true, too.

You can use the * prefix to pass list-like objects to a function that specifies its arguments.

```
def arg_expansion_example(x, y):
    return x**y
```

```
my_args = [2, 8]
arg_expansion_example(*my_args)
```

But, the passed object must be the right length.

```
my_args2 = [2, 8, 5]
arg_expansion_example(*my_args2)
```

```
## **my_dict
```

67.8 Default Arguments

Use default arguments to set the value of arguments when left unspecified.

```
def show_results(precision, printing=True):
    precision = round(precision, 2)
    if printing:
        print('precision =', precision)
    return precision
```

```
pr = 0.912
res = show_results(pr)
```

```
precision = 0.91
```

The function call didn't specify `printing`, so it defaulted to True.

NOTE: Default arguments must follow non-default arguments. This causes trouble:

```
def show_results(precision, printing=True, uhoh):
    precision = round(precision, 2)
    if printing:
      print('precision =', precision)
    return precision
```

```
SyntaxError: non-default argument follows default argument (<ipython-input-19-29f5905a75a5>,
```

67.9 Returning Values

Functions are not required to have return statement.

If there is no return statement, a function returns `None`.

Functions can return no value (`None`), one value, or many.

Many values are returned as a tuple.

Any Python object can be returned.

```
## returns None, and prints.

def fcn_nothing_to_return(x, y):
    out = 'nothing to see here!'
    print(out)
```

```
fcn_nothing_to_return(x, y)
```

```
nothing to see here!
```

```
r = fcn_nothing_to_return(1, 1)
print(r)
```

```
nothing to see here!
None
```

```
## returns three values

def negate_coords(x, y, z):
    return -x, -y, -z
```

```
a, b, c = negate_coords(10, 20, 30)
print('a =', a)
print('b =', b)
print('c =', c)
```

```
a = -10
b = -20
c = -30
```

```
foo = negate_coords(10, 20, 30)
```

```
foo, len(foo)
```

```
((-10, -20, -30), 3)
```

If you don't need an output, use the dummy variable _

```
d, e, _ = negate_coords(10,20,30)
print('d =', d)
print('e =', e)
```

Note: It's generally a good idea to include return statements, even if not returning a value.

This shows that you did not forget to consider the return value.

You can use `return` or `return None`.

Functions can contain multiple return statements.

These may be used under different logical conditions.

```python
def absolute_value(num):
    if num >= 0:
        return num
    return -num
```

```python
absolute_value(-4)
```

```python
absolute_value(4)
```

For non-negative values, the first `return` is reached.
For negative values, the second `return` is reached.

67.10 Function Design

A function is not just a bag of code!

Some good practices for creating and using functions:

- design a function to do one thing

Make them as simple as possible, which makes them:

- more comprehensible
- easier to maintain
- reusable

This helps avoid situations where a team has 20 variations of similar functions.

Give your function a good name.

- It should reflect the action it performs.
- Be consistent in your naming conventions.
- A name like `compute_variances_sort_save_print` suggests the function is overworked!

If the function `compute_variances` also produces plots and updates variables, it will cause confusion.

Always give your function a docstring - Particularly important since indicating data types is not required.
- As a side note, you can include this information by using **type annotation**.

Finally, at some point you may be interested to learn some of the formatting languages that have been developed to write docstrings. See Lutz 2019 and this web page about Documenting Python Code for more info.

68 NB: Importing Functions

68.1 Importing

Calling a function from the “math” library is straightforward:

1. Import Python’s Math library with the command `import math`
2. Call methods from the imported `math` object using “dot” notation, that is, .(any parameters).

For example:

```
math.sqrt(12)
```

Put all of your import statements at the very top of your code, before anything else, other than any header comments (which you should have).

Here are some example math functions:

```
import math # Typically best to put this line of code at the TOP of the file
```

```
math.sqrt(12)
```

```
3.4641016151377544
```

```
math.floor(2.5) # returns largest whole number less than the argument
```

```
2
```

Here’s an example using the random library (a class).

```
import random # Typically best to put this line of code at the TOP of the file
```

```
random.random()# will return a number between 0 and 1
```

```
0.3599068479674543
```

```
random.randint(1, 100) # this will return a random integer in the range 1-100
```

```
18
```

68.2 Importing Specific Functions

If you know what specifics function you are going to use from a library, you can import them directly, like so:

```
from math import sqrt
```

This has two effects: 1. It reduces the memory used by the library in your program. 2. It allows you to call the function directly, with the object dot notation.

```
from math import sqrt
```

```
sqrt(99)
```

```
9.9498743710662
```

68.3 Aliasing

To avoid having the function name conflict with an existing function in your program, you can alias the imported function like so:

```
from math import sqrt as SquareRoot
```

```
SquareRoot(65000)
```

```
254.95097567963924
```

69 Extra

```
def square(number):
    return number * number  # square a number

def addTen(number):
    return number + 10  # Add 10 to the number

def numVowels(string):
    string = string.lower()  # convert user input to lowercase
    count = 0
    for i in range(len(string)):
        if string[i] == "a" or string[i] == "e" or \
           string[i] == "i" or string[i] == "o" or \
           string[i] == "u":
           count += 1 # increment count
    return count
```

70 NB: Lambda Functions

70.1 Introduction

Python lambda functions are small, informal functions. They don't get a name.

The are "anonymous."

From Lutz 2019:

> Besides the `def` statement, Python also provides an expression form that generates function objects. Because of its similarity to a tool in the Lisp language, it's called lambda. Like `def`, this expression creates a function to be called later, but it returns the function instead of assigning it to a name. This is why lambdas are sometimes known as anonymous (i.e., unnamed) functions. In practice, they are often used as a way to inline a function definition, or to defer execution of a piece of code.

The general form of a lambda function is:

```
lambda x: x
```

```
<function __main__.<lambda>(x)>
```

You can call the function like this:

```
(lambda x: x)(2)
```

```
2
```

increment x

```
(lambda x: x+1)(5)
```

```
6
```

sum two variables

```
lambda x, y: x + y
```

```
<function __main__.<lambda>(x, y)>
```

70.2 Assigned to a Variable

Even though they don't get a name, they can be assigned to variables.

Here, a lambda function gets assigned to `sum_two_vars`.

```
sum_two_vars = lambda x, y: x + y
```

```
sum_two_vars(2,4)
```

```
6
```

Check if a value is non-negative

```
is_non_negative = lambda x: x >= 0
```

```
is_non_negative(-9)
```

```
False
```

```
is_non_negative(0)
```

```
True
```

Package first element and all data into tuple

```
pack_first_all = lambda x: (x[0], x)
```

```
casado = ('rice','beans','salad','plaintain','chicken') # a typical Costa Rican dish

pack_first_all(casado)
```

```
('rice', ('rice', 'beans', 'salad', 'plaintain', 'chicken'))
```

Check for keyword "dirty"

```
is_dirty = lambda txt: 'dirty' in txt
```

```
kitchen_inspection = 'dirty dishes'
is_dirty(kitchen_inspection)
```

```
True
```

```
kitchen_inspection = 'pretty clean!'
is_dirty(kitchen_inspection)
```

```
False
```

pass *args for unspecified number of arguments

```
(lambda *args: sum(args))(1,2,3)
```

```
6
```

70.3 Using Lambda

Lambda functions are often used in Pandas. We will discuss there use in more detail when we get to that topic.

71 NB: Recursion

Concepts - recursion - recursive function - stack - stack overflow

71.1 Introduction

A recursive function is **a function that calls itself**.

This is weird, since it does not seem possible. How can a definition refer to itself?

In philosophy, this is expressed in the Barber's Paradox:

> The barber is the one who shaves all those, and those only, who do not shave themselves. Does the barber shave himself?

Formally, it is a type of self-reference, like `This sentence is false.`

A Cute Definition

recursion - the art of defining something (at least partly) in terms of itself, which is a naughty no-no in dictionaries but often works out okay in computer programs if you're careful not to recurse forever (which is like an infinite loop with more spectacular failure modes).

Source: *PerlDoc*

71.1.1 A Formal Definition

In mathematics and computer science, a class of objects or methods exhibits *recursive behavior* when it can be defined by two properties:

A **simple base** case (or cases): a terminating scenario that does not use recursion to produce an answer.

A **recursive step**: a set of rules that reduces all successive cases toward the base case.

71.1.2 As Seen in Nature

Recursion occurs naturally when a process applies a rule to itself successively.

We see this in fractals.

71.1.3 Infinite Loops and Stack Overflows

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

The **call stack** is where information is stored relating to the active subroutines in a program.

The call stack has a limited amount of available memory. When excessive memory consumption occurs on the call stack, it results in a **stack overflow error**.

71.1.4 A Note of Caution

So, Recursion is cool, but is expensive and complicated.

Recursive functions can usually be implemented by traditional loops.

71.2 Example: Computing Factorials

Source

The factorial of a number n is the product of all the integers from 1 to n.

For example, the factorial of 5 (denoted as 5!) is $1 \times 2 \times 3 \times 4 \times 5 = 120$.

Let's implement this in code using a recursive function.

71.2.1 Recursive Function

```
n = 5
```

```
##| tags: []
def factorial(x):
    "Finds the factorial of an integer using recursion"
    if x == 1: # Base condition
        return 1
    else:
        return x * factorial(x-1)
```

```
##| tags: []
%time factorial(n)
```

```
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 7.87 µs
```

```
120
```

71.2.2 As a while loop

```
def factorial_while(x):
    "Finds the factorial of an integer using a while loop"
    f = x
    while x > 1:
        x -= 1
        f *= x
    return f
```

```
%time factorial_while(n)
```

```
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 6.44 µs
```

```
120
```

71.2.3 As a for loop

```
def factorial_for(x):
    "Finds the factorial of an integer using a for loop"
    f = x
    for i in range(1, x):
        x -= 1
        f *= x
    return f
```

```
%time factorial_for(n)
```

```
CPU times: user 4 µs, sys: 0 ns, total: 4 µs
Wall time: 7.15 µs
```

```
120
```

71.2.4 Compare functions as n increases

71.2.4.1 Increase n to 50

```
n = 50
%time factorial(n)
```

```
CPU times: user 30 µs, sys: 0 ns, total: 30 µs
Wall time: 33.4 µs
```

```
30414093201713378043612608166064768844377641568960512000000000000
```

```
%time factorial_while(n)
%time factorial_for(n)
```

```
CPU times: user 7 µs, sys: 1 µs, total: 8 µs
Wall time: 10.7 µs
CPU times: user 7 µs, sys: 0 ns, total: 7 µs
Wall time: 9.06 µs
```

```
30414093201713378043612608166064768844377641568960512000000000000
```

71.2.4.2 Increase n to 500

```
n = 500
```

```
%time factorial(n)
```

```
CPU times: user 494 µs, sys: 0 ns, total: 494 µs
Wall time: 499 µs
```

```
12201368259911100687012387854230469262535743428031928421924135883858453731538819976054964475
```

```
%time factorial_while(n)
```

```
CPU times: user 85 µs, sys: 5 µs, total: 90 µs
Wall time: 93 µs
```

```
12201368259911100687012387854230469262535743428031928421924135883858453731538819976054964475
```

```
%time factorial_for(n)
```

```
CPU times: user 88 µs, sys: 0 ns, total: 88 µs
Wall time: 90.8 µs
```

```
12201368259911100687012387854230469262535743428031928421924135883858453731538819976054964475
```

71.2.4.3 Increase n to 5000

```
n = 5000
%time factorial(n)
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

```
factorial_while(n)
```

```
42285779266055435222010642002335844053907866746266467488497824021813580527081082006908990478
```

```
%time factorial_while(n)
```

```
CPU times: user 4.93 ms, sys: 0 ns, total: 4.93 ms
Wall time: 4.94 ms
```

```
42285779266055435222010642002335844053907866746266467488497824021813580527081082006908990478
```