DS 5001: Programming for Data Science

Spring 2023

Rafael C. Alvarado

5/8/21

Table of contents

Part I

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

ℹ 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.

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.

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.

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(), 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 semester, 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.

When submitting homework assignments, don't forget to write the assignment title, your name, your UVA computing ID, and date at the top of each assignment.

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 practice scripts and notebooks made available to you. These are designed to exemplify the concepts conveyed in reading and dialog. 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.

Quizzes typically have ten questions and are worth 10 points each.

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.

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 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 forthcoming.

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

1.4.2 Scale

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.

Other Texts

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.

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 Resources

Students have three choices for participating in coding exercises and completing coding assignments in the course:

Rivanna, UVA's High Performance Computing Cluster. This may be access through the web or by SSH (don't worry if you don't know what that means yet). More information about this resource may be found here.

CEDS, the Computational Environment for Data Science. This is a web-hosted and Windows-based virtual environment. You can access the site here. Here are step-by-step instructions for how to access this service.

Your own computer. Please note that if you choose this option it will be harder to trouble-shoot any system level problems t hat may arise.

1.7 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.

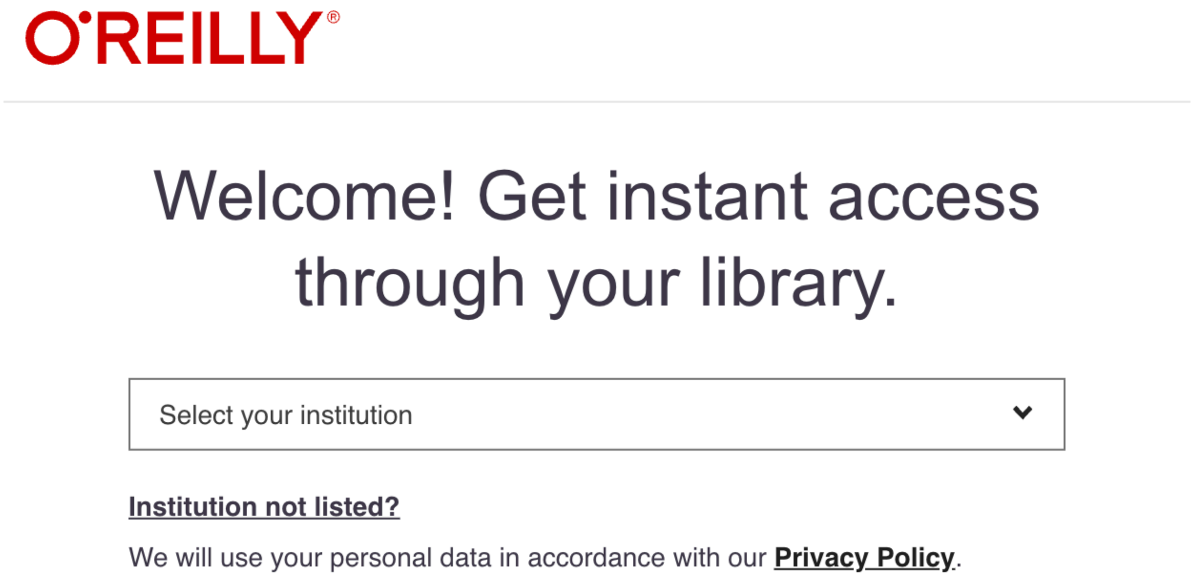
2 Setting Up O'Reilly

O'Reilly for Higher Education contains books on all aspects of computers, programming and the Web from such publishers as O'Reilly, Sams, New Riders, Adobe. Students can access content for free.

Follow these instructions to set up your account:

Go to https://www.oreilly.com/library/view/temporary-access/.

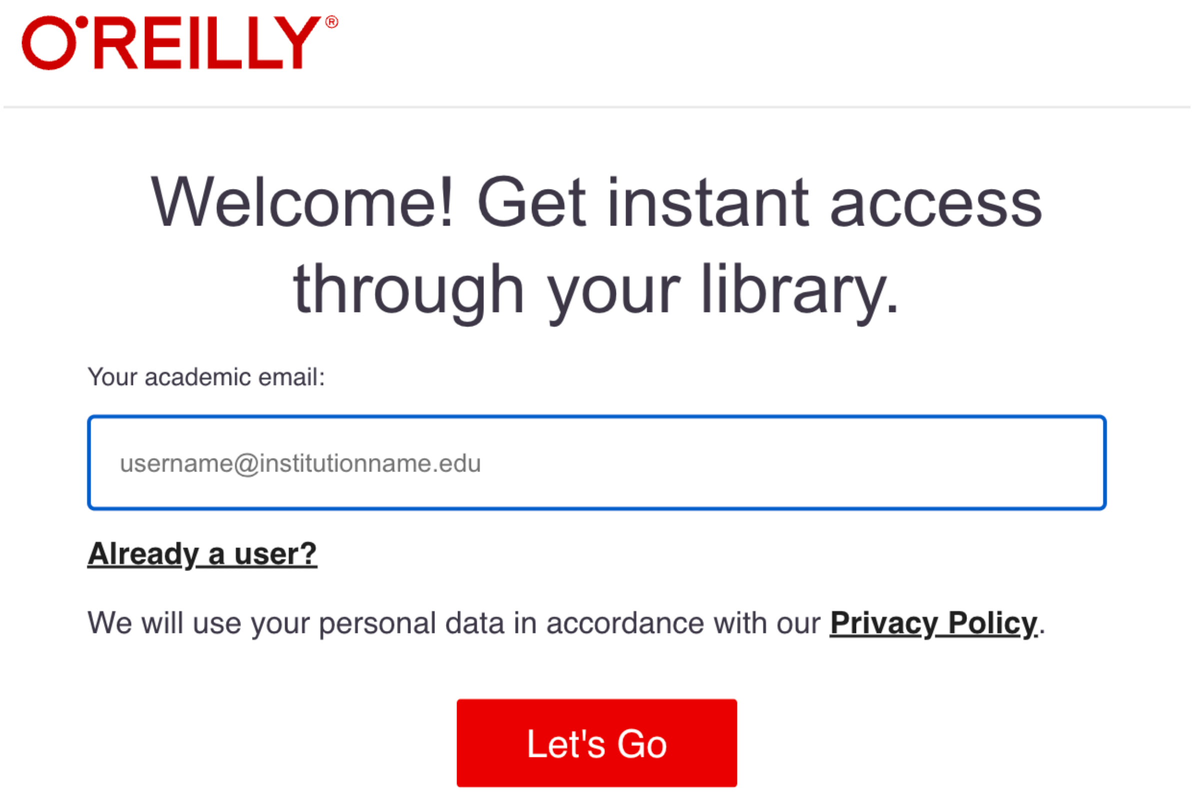
You should see this dialog box:



Welcome! Get instant access through your library.

We will use your personal data in accordance with our **Privacy Policy**.

Click on Institution not listed? This will produce the following dialog box:



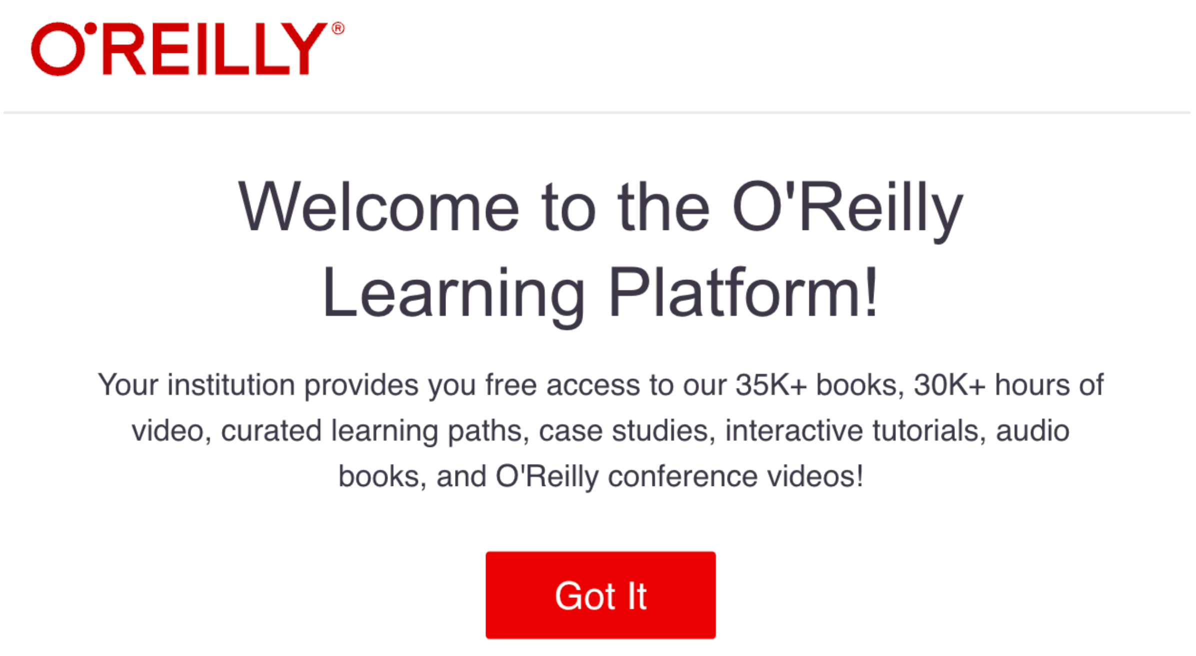
Welcome! Get instant access through your library.

Your academic email:

username@institutionname.edu

We will use your personal data in accordance with our **Privacy Policy**.

If you are first time user, enter your UVA email address and clicl on the big red button. This should produce the following message:


Welcome to the O'Reilly Learning Platform!

Your institution provides you free access to our 35K+ books, 30K+ hours of video, curated learning paths, case studies, interactive tutorials, audio books, and O'Reilly conference videos!

Now you can access all the books in the collection. When you return, click on Already a user? in the second dialog box.

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

3 About Rivanna

3.1 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.)

3.2 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.

3.3 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.

3.4 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

4 Using Unix

4.1 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.)

4.2 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

4.3 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:

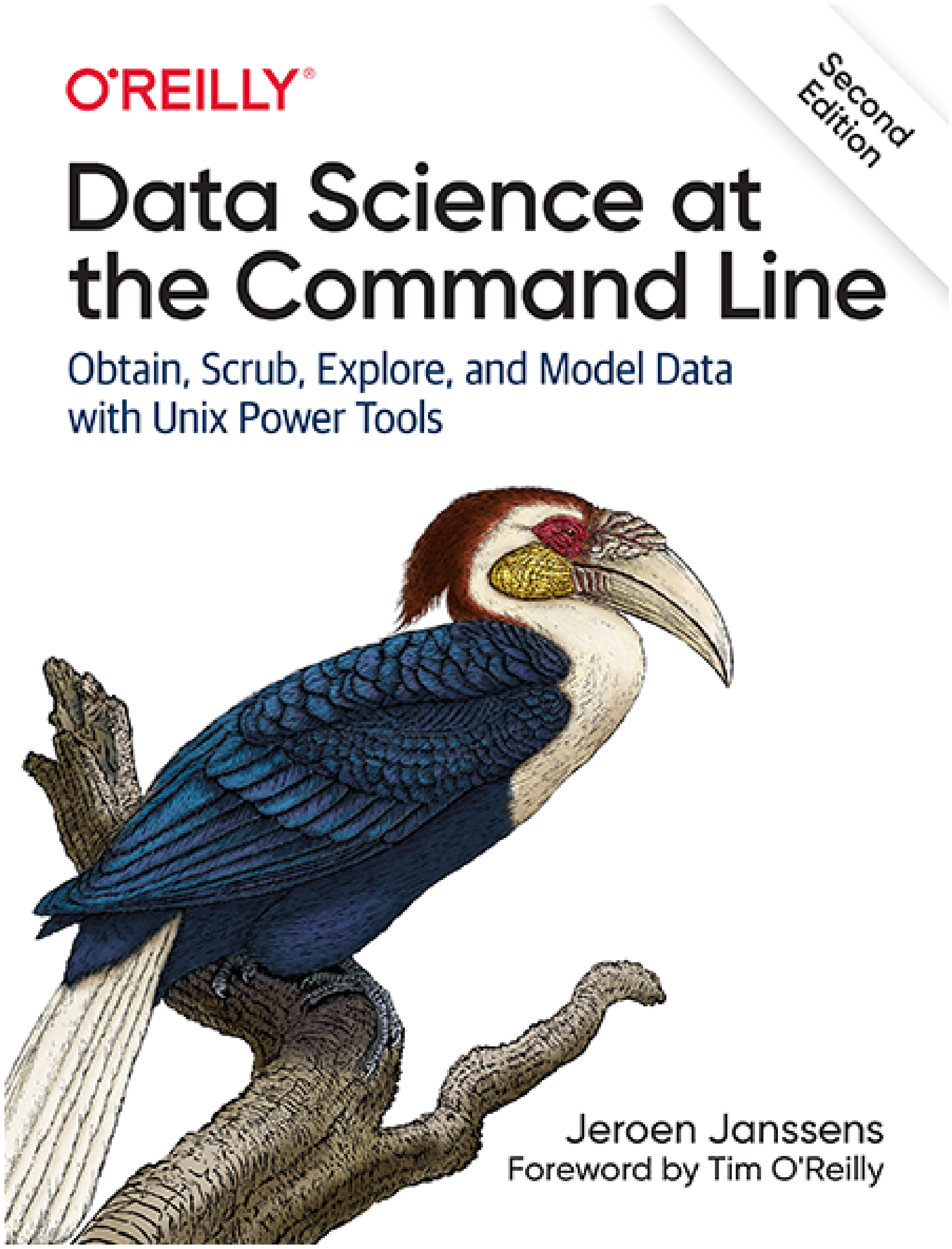
- `cat` – concatenate files and show contents to the standard output
- `more` – basic pagination when viewing text files or parsing Unix commands output
- `less` – an improved pagination tool for viewing text files (better than more command)
- `head` – show the first 10 lines of text file (you can specify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

4.4 Miscellaneous

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

4.5 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*.

5 SSH for GitHub

5.1 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.

5.2 Steps

5.2.1 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).

5.2.2 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.

5.3 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

6 Git and GitHub

6.1 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.

6.2 Using Git and GitHub Together

Source

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

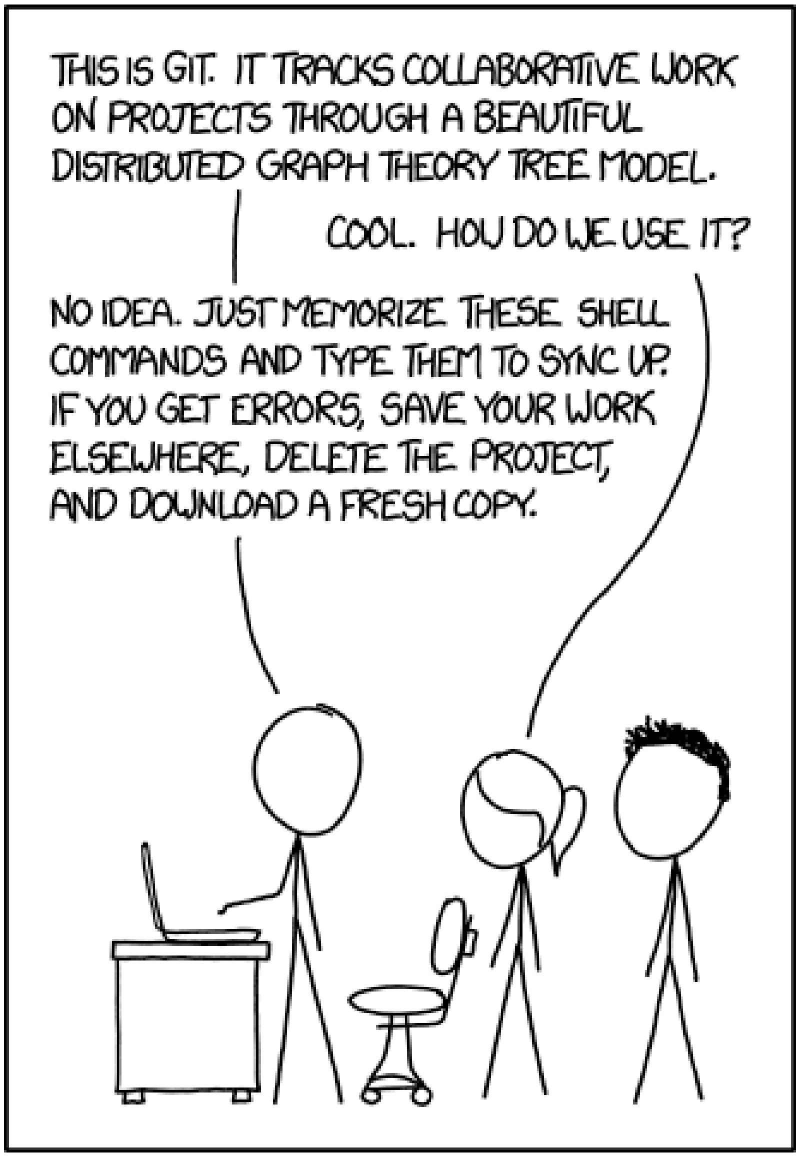

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Figure 6.1: 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.

ℹ 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:

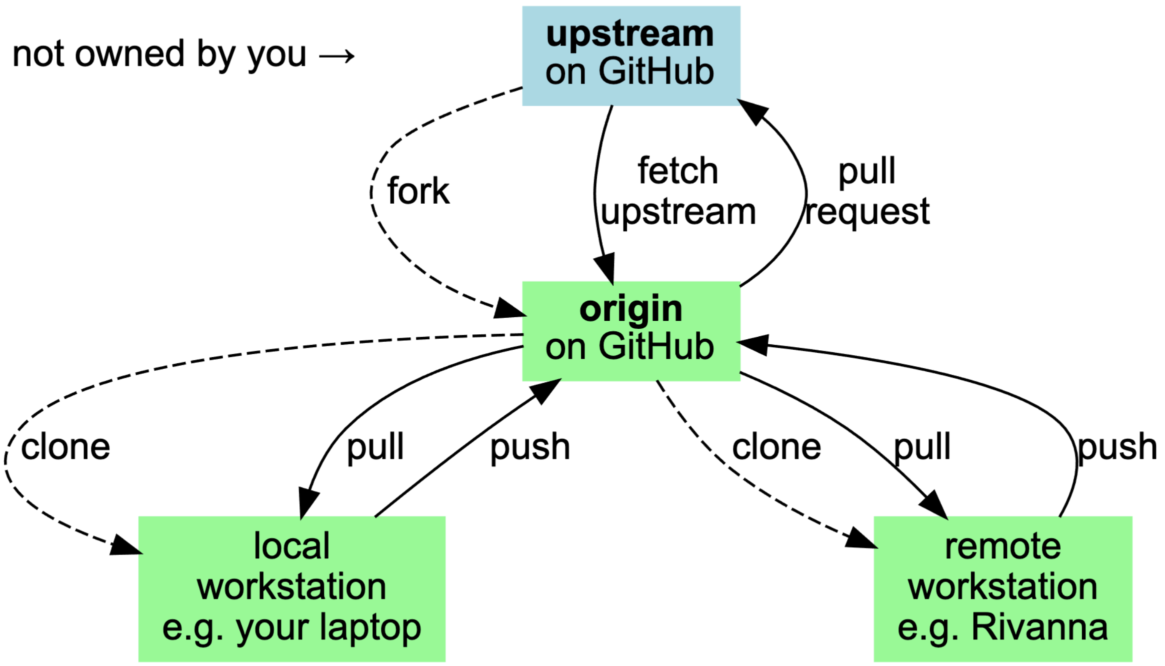


Figure 6.2: Diagram of common git workflow

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.

6.3 To Learn More

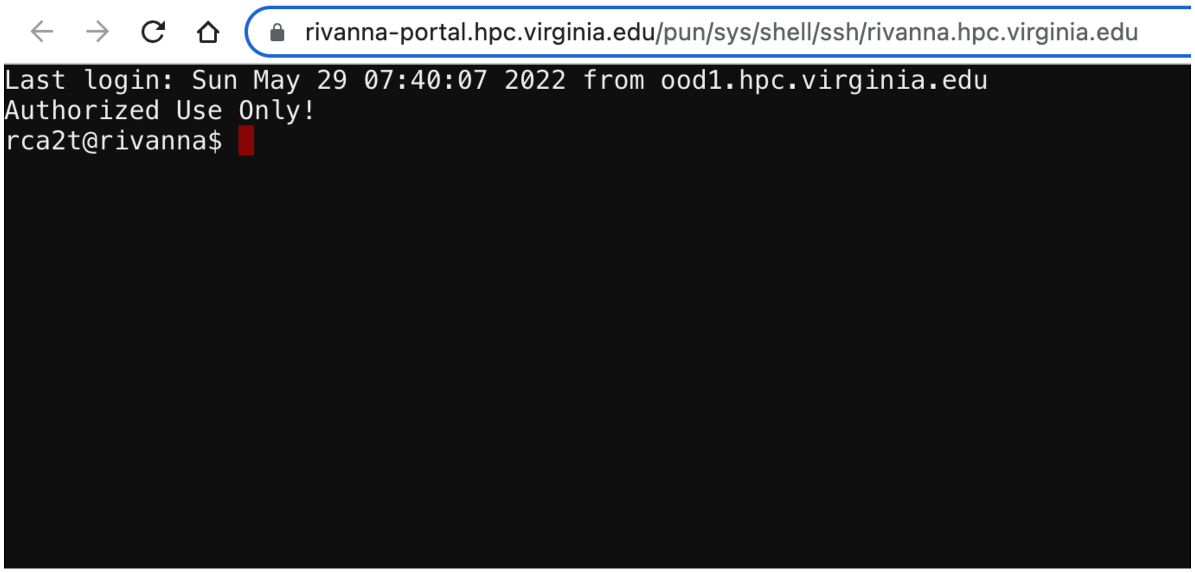
- Videos
- Book

7 Activity: Using Rivanna

After reading the previous documents on Rivanna and Unix, try to complete this activity before class.

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 7.1: 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.

8 Activity: Using Git and GitHub

During the technical orientation, you set up a GitHub account.

You also spent a little time browsing a sample repository, which you may wish to revisit:

- https://github.com/UVADS/orientation-technical

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 we created earlier.

Also, we assume you have already created a GitHub account. å ***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.

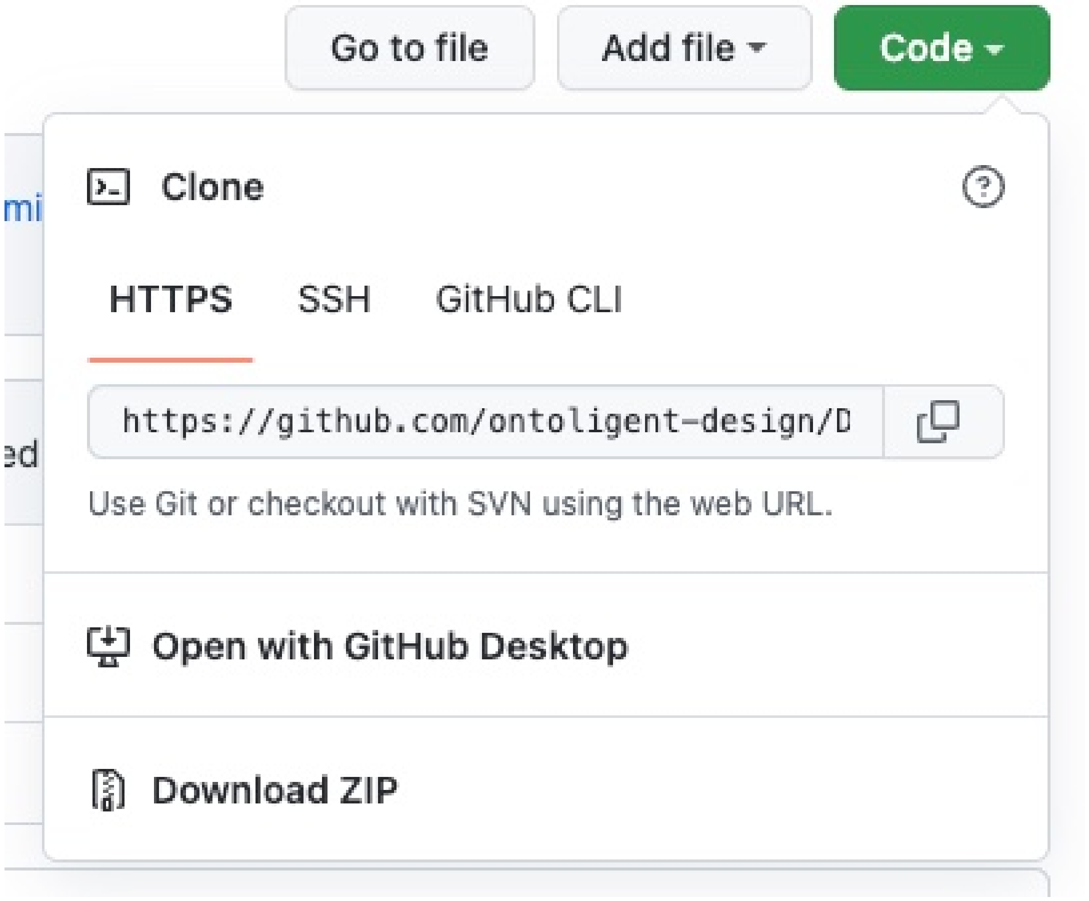
Click on the Fork icon in the upper right and follow the prompts to finish the process.

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:


Go to file
Add file
Code
Clone
HTTPS
SSH
GitHub CLI
https://github.com/ontoligent-design/D
Use Git or checkout with SVN using the web URL.
Open with GitHub Desktop
Download ZIP

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 `lessons/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/lessons/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 8.1: 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.

8.1 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.
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)

9 Data and Code

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.

9.1 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.

9.2 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.

10 Python Object Types

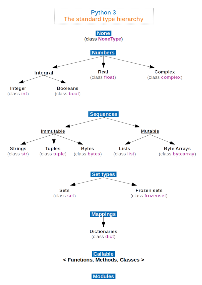
Python is organized into a hierarchy of object types. Sometimes, these are just call **types**.

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.



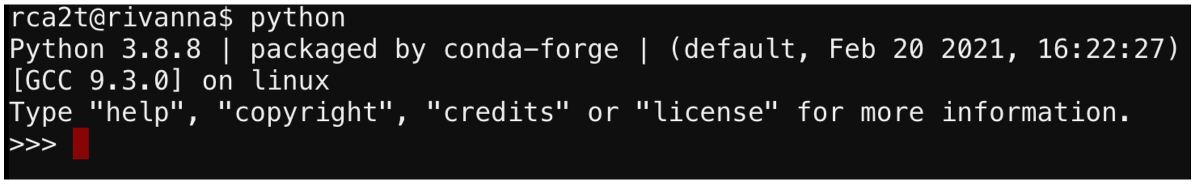
11 Activity: Hello, World!

11.1 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 :-)

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

11.2 Python Files

Now create a file called `hello.py` using the command line editor `nano`.

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

12 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.

Remember to select just the resources needed and to enter our course allocations (`msds_ds5100`).

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`.

13 NB: Data Types, Operators, and Expressions

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

```
10
```

```
10
```

Floats (decimals)

```
3.14
```

```
3.14
```

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'
```

```
'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)
```

14 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'>
```

15 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'>
```

16 Deleting variables with `del()`

```python
x = 101.25
```

```python
x
```

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

```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)
```

17 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
```

18 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

18.1 `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'
```

18.2 `ord()`

Converting a character to it’s code point

```python
ord?
```

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

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

```
(97, 65)
```

19 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.

19.1 Arithmetic Operators

19.1.1 floor division //

```
5 // 2
```

```
2
```

```
-5 // 2
```

```
-3
```

```
5.5 // 2
```

```
2.0
```

19.1.2 modulus %

Returns the remainder

```python
5 % 2
```

```
1
```

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

Look at this …

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

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

19.1.3 exponentiation **

```python
5**3
```

```
125
```

19.2 String Operators

19.2.1 concatenation +

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

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

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

```python
my2ndString
```

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

19.2.2 repetition *

```
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 :-)

19.3 Assignment Operator =

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

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

```
epoch: 20
```

19.4 Comparison Operators

Comparisons are questions.

They return a boolean value.

19.4.1 equality ==

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

```
True
```

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

```
False
```

Can we compare strings

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

```
True
```

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

```
(65, 66)
```

19.4.2 inequality !=

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

```
True
```

19.5 Logical Operators

Python uses words where other languages will use other symbols.

19.5.1 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
```

19.5.2 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: SyntaxWarni
  x is not 'fail'
```

```
False
```

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

```
True
```

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

```
False
```

19.5.3 Negation `not`

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

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

20 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
```

21 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
```

22 NB: Numbers

These are built-in mathematical functions for numbers.

22.1 `pow()` Power

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

22.2 `abs()` Absolute value

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

22.3 `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
```

23 Math library functions

See the Python docs on the math library.

```
import math
```

23.1 `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
```

23.2 `math.log()`

```
math.log?
```

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

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

24 The Random library

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

```
import random
```

24.1 `random.random()`

```
random.random?
```

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

24.2 `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
```

25 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

```python
cache = True

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

```
data will be cached
```

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

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

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

We'll cover these later.

```python
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.

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

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

26 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!"
```

26.1 Quote prefixes

26.1.1 `r` strings

Prefixing a string causes escape characters to be uninterpreted.

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

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

26.1.2 `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.")
```

27 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)")
```

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

```
# 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!')
```

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")
```

28 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:

```
'''
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.

29 Run-time User Input

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

30 Some String Functions

```
Built-in string methods and functions.

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

30.1 `.lower()`, `.upper()`

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

30.2 `.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('.')
```

```
montyPythonQuote
```

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

30.3 `.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')
```

30.4 `.replace()`

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

30.5 `.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))
```

30.6 `.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
```

31 Strings are Lists

Actually, they are list-like.

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

31.1 `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)
```

31.1.1 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.
```

31.1.2 Slicing

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

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

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

```
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**.

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

Add strings and handle pathing

32 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.

32.1 Constructing

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

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

32.2 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)
```

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

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

32.3 Slicing

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

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

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

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

32.4 Multiply lists by a scalar

A scalar is a single value number.

```
numbers * 2
```

32.5 Concatenate lists with +

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

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

32.6 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
```

32.7 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]
```

32.8 Lists can concatenated with +

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

```
variables + response
```

33 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.

33.1 Constructing

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

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

```
dict2
```

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

```
dict3
```

33.2 Retrieve a value

Just write a key as the *index*.

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

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

33.3 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
```

34 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
```

34.1 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
```

35 Common functions and methods to all sequences

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

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

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

35.1 Membership with `in`

Returns a boolean.

```
'Sam' in phonelist
```

36 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)
```

37 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)
```

38 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
```

38.1 Try with `dict`

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

38.2 Use `try` and `except`

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

```
words
```

38.3 Or use `.get()`

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

38.4 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)

39 NB: Control Structures

Topics:

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

39.1 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)

39.2 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."

39.3 Conditions

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

```
val = -2

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

39.3.2 `elif`

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

```
val = -2

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

39.3.3 `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')
```

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

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

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

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

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

39.3.5 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')
```

```
## incorrect - misplaced parenthesis

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

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?

39.4 Loops

39.4.1 `while`

What does this print?

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

39.4.2 `break` to exit the loop altogether

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

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

The `break` causes the loop to end early

39.4.3 `continue` to stop the current iteration

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

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

The `continue` causes the loop to skip printing 4

39.4.4 `for`

iterate over an iterable

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

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

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)
```

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)
```

39.5 `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:

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

40 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

41 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.

42 Lists

42.1 iterating using `for`

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

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

```
living room
was
quite
large
```

42.2 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:

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

```
StopIteration:
```

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

```
type(myit)
```

```
list_iterator
```

```
help(myit)
```

```
Help on list_iterator object:

class list_iterator(object)
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __length_hint__(...)
 |      Private method returning an estimate of len(list(it)).
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(...)
 |      Return state information for pickling.
 |
 |  __setstate__(...)
 |      Set state information for unpickling.
```

```
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.

43 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

44 Sets

iterating using `for`

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

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

```
cinderella
belle
rapunzel
```

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

```
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
```

45 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
```

46 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
```

47 Dictionaries

iterating using `for`

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

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

```
fall
spring
```

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

```
fall
spring
```

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

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

```
# 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
```

48 Ranges

iterating using `for`

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

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

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

49 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
```

50 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.

```python
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()`

51 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

52 List Comprehensions

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

Without list comprehensions, you might do this:

52.1 Check if Odd

```python
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:

```python
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.

53 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.

54 More examples

54.1 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'}
```

54.2 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']
```

55 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)

56 NB: Importing Functions

56.1 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
```

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

```
18
```

56.2 Importing Specific Functions

If you know what specifics function you are going to use from a library, you can import them directly, like so:

```python
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.

```python
from math import sqrt
```

```python
sqrt(99)
```

```
9.9498743710662
```

56.3 Aliasing

To avoid having the function name conflict with an existing function in your program, you can alias the imported function like so:

```python
from math import sqrt as SquareRoot
```

```python
SquareRoot(65000)
```

```
254.95097567963924
```

```python
def square(number):
    return number * number  # square a number
```

```python
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
```

57 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

57.1 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".

57.2 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.

57.3 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)
```

57.4 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

57.5 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.

57.5.1 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.

57.5.2 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)
```

```
## 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.

57.5.3 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
```

57.6 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)
```

57.6.1 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)
```

```
fcn_swapped_args(y, x)
```

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)
```

57.6.2 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.

57.7 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")
```

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)
```

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
```

57.8 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)
```

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
```

57.9 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)
```

```
r = fcn_nothing_to_return(1, 1)
print(r)
```

```
## 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)
```

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.

```
def absolute_value(num):
    if num >= 0:
        return num
    return -num
```

```
absolute_value(-4)
```

```
absolute_value(4)
```

For non-negative values, the first `return` is reached.
For negative values, the second `return` is reached.

57.10 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.

58 NB: Lambda Functions

58.1 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:

```python
lambda x: x
```

```
<function __main__.<lambda>(x)>
```

You can call the function like this:

```python
(lambda x: x)(2)
```

```
2
```

increment x

```python
(lambda x: x+1)(5)
```

```
6
```

sum two variables

```
lambda x, y: x + y
```

```
<function __main__.<lambda>(x, y)>
```

58.2 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)
```

Check if a value is non-negative

```
is_non_negative = lambda x: x >= 0
```

```
is_non_negative(-9)
```

```
is_non_negative(0)
```

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)
```

Check for keyword "dirty"

```
is_dirty = lambda txt: 'dirty' in txt
```

```
kitchen_inspection = 'dirty dishes'
is_dirty(kitchen_inspection)
```

```
kitchen_inspection = 'pretty clean!'
is_dirty(kitchen_inspection)
```

pass *args for unspecified number of arguments

```
(lambda *args: sum(args))(1,2,3)
```

58.3 Using Lambda

Lambda functions are often used in Pandas. We will discuss there use in more detail when we get to that topic.

59 NB: Recursion

Concepts - recursion - recursive function - stack - stack overflow

59.1 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*

59.1.1 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.

59.1.2 As Seen in Nature

Recursion occurs naturally when a process applies a rule to itself successively.

We see this in fractals.

59.1.3 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**.

59.1.4 A Note of Caution

So, Recursion is cool, but is expensive and complicated.

Recursive functions can usually be implemented by traditional loops.

59.2 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.

59.2.1 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)
```

59.2.2 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)
```

59.2.3 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)
```

59.2.4 Compare functions as n increases

59.2.4.1 Increase n to 50

```
n = 50
%time factorial(n)
```

```
%time factorial_while(n)
%time factorial_for(n)
```

59.2.4.2 Increase n to 500

```
n = 500
```

```
%time factorial(n)
```

```
%time factorial_while(n)
```

```
%time factorial_for(n)
```

59.2.4.3 Increase n to 5000

```
n = 5000
%time factorial(n)
```

```
factorial_while(n)
```

```
%time factorial_while(n)
```

```
%time factorial_for(n)
```

59.3 Example: The Fibonacci sequence

Fib(0) = 0 (base case 1)

Fib(1) = 1 (base case 2)

For all integers n > 1, Fib(n) = Fib(n − 1) + Fib(n − 2)

```
def Fibonacci(n):
    "Compute a Fibonacci Sequence using recursion"

    # If n is negative
    if n < 0:
        print("Incorrect input. Value must be 0 or greater.")

    # If n is 0
    elif n == 0:
        return 0

    # If n is 1 or 2
    elif n == 1 or n == 2:
        return 1

    else:
        return Fibonacci(n - 1) + Fibonacci(n - 2)
```

```
n = 9
```

```
Fibonacci(9)
```

```
for n in range(100):
    if n > 0: print(", ", end="")
    print(Fibonacci(n), end="")
```

59.3.1 As a for loop

```
def fibber(r:int = 10):
    """
    Computes a Fibonacci Sequence using a for loop.
    Parameter r must be in integer > 3. Defaults to 10.
    Returns a string as a comma-limited series.
    """
    seq = [1,1,2]
    kernel = lambda x, i: x[i-1] + x[i-2]
    for n in range(3, r):
        seq.append(seq[n-1] + seq[n-2])
    return ', '.join([str(x) for x in seq])
```

```
fibber(20)
```

59.4 Aside: A General Sequence Function

Recursive functions are often used to produce mathematical sequences, but since they have limits on depth, they are of limited use for this purpose.

Here is a function that can combine many sequences using two sequence parameters: * The initial state of the sequence, represented as the list `seq`. * For example, in the Fibonacci sequence, seq is `[1, 1, 2]` * The function to apply to the sequence at each iteration, represneted as a `lambda` function with the arguments `x` and `i` for the the sequence list `seq` and the iteration number respectively. * For example, in the Fibonacci sequence the kernel function is `lambda x, i: x[i-1] + x[i-2]`

```
##| tags: []
def sequencer(n:int = 10, seq=[1, 1, 2], kernel=lambda x, i: x[i-1] + x[i-2]):
    """
    Computes a Sequence using a for loop.

    Parameter n in integer which must be > 3. Defaults to 10.
    Parameter seq is as list in the initial state of the sequence. Must have at least one
    Parameter kernel is the kernel function applied to the series at each iteration. x sta

    Returns a string as a comma-limited series.
    """

    for i in range(len(seq), n): seq.append(kernel(seq, i))
    return ', '.join([str(x) for x in seq])
```

```
n = 8
```

```
%time sequencer(n, [0], lambda x, i: i)
```

The series of positive integers

```
sequencer(n, [1], lambda x, i: x[i-1] + 1)
```

The series of even numbers

```
sequencer(n, [2], lambda x, i: x[i-1] + 2)
```

The series of odd numbers

```
sequencer(n, [1], lambda x, i: x[i-1] + 2)
```

The series of Fibonacci numbers

```
sequencer(n, [1,1,2], lambda x, i: x[i-1] + x[i-2])
```

The series of Squares

```
sequencer(n, [2], lambda x, i: x[i-1]**2)
```

60 NB: Variable Scope

60.1 Overview

A variable's **scope** is the part of a program where it is **visible**.

- Scope refers to the **coding region**, such as a function block, from which a particular Python object is accessible.
- Visible means available or **usable** to the code block in question.
- If a variable is **in scope** to a function, it is visible the function.
- If it is **out of scope** to a function, it is not visible the function.

When a variable is defined **inside** of a function, is is not visible outside of the function. * We say such variables are **local** to the function. * They are also removed from memory when the function completes.

When a variable is defined **outside** of any function in a script, it is visible to any function inside of the script * We say such variables are **global** to the functions in the file or context in which the variables are defined. * A function can replace a global variable with local variable by defining that variable. In this case, a variable can have global and local versions in the same program.

Sometimes variable scope is called **lexical** scope.

It is helpful to have a good understanding of scope to avoid surprises and confusion.

The concept is easier than it may look in the abstract. Let's look at some examples where we vary the use of local and global definitions of `x`.

60.2 Lutz on Scoping

Here's an excerpt from Lutz, Chapter 17. Please read the whole thing.

The enclosing module is a global scope. Each module is a global scope—that is, a namespace in which variables created (assigned) at the top level of the module file live. Global variables become attributes of a module object to the outside world after imports but can also be used as simple variables within the module file itself.

The global scope spans a single file only. Don't be fooled by the word "global" here—names at the top level of a file are global to code within that single file only. There is really no notion of a single, all-encompassing global file-based scope in Python. Instead, names are partitioned into modules, and you must always import a module explicitly if you want to be able to use the names its file defines. **When you hear "global" in Python, think "module."**

Assigned names are local unless declared global or nonlocal. By default, all the names assigned inside a function definition are put in the local scope (the namespace associated with the function call). If you need to assign a name that lives at the top level of the module enclosing the function, you can do so by declaring it in a global statement inside the function. If you need to assign a name that lives in an enclosing def, as of Python 3.X you can do so by declaring it in a nonlocal statement.

All other names are enclosing function locals, globals, or built-ins. Names not assigned a value in the function definition are assumed to be enclosing scope locals, defined in a physically surrounding def statement; globals that live in the enclosing module's namespace; or built-ins in the predefined built-ins module Python provides.

Each call to a function creates a new local scope. Every time you call a function, you create a new local scope—that is, a namespace in which the names created inside that function will usually live. You can think of each def statement (and lambda expression) as defining a new local scope, but the local scope actually corresponds to a function call. Because Python allows functions to call themselves to loop—an advanced technique known as recursion and noted briefly in Chapter 9 when we explored comparisons—each active call receives its own copy of the function's local variables. Recursion is useful in functions we write as well, to process structures whose shapes can't be predicted ahead of time; we'll explore it more fully in Chapter 19.

And here is a nice visualization of scopes:

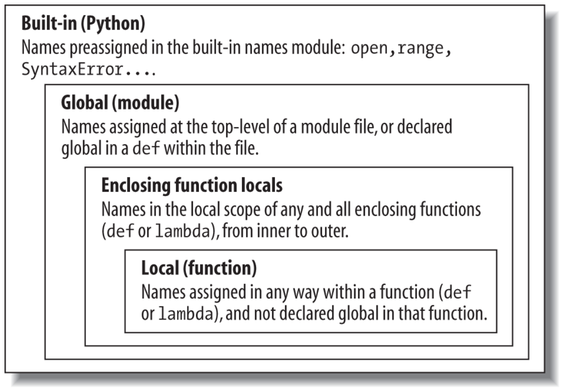

Built-in (Python)
Names preassigned in the built-in names module: open,range, SyntaxError....
Global (module)
Names assigned at the top-level of a module file, or declared global in a def within the file.
Enclosing function locals
Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.
Local (function)
Names assigned in any way within a function (def or lambda), and not declared global in that function.

Figure 60.1: Car with tinted glass

Please read Lutz, Chapter 17 for a good overview of scoping.

60.3 The Tinted Glass Metaphor

So, code regions within a program are like vehicles with tinted glass: * Passengers can see outside, but outsiders can't see inside. * Passengers in a vehicle can let outsiders look in by lowering the window.

Or something. Lile models, all metaphors are wrong, but some are useful. :-)

60.4 Example 1

x defined outside a function but used inside of it

In the code below: * `x` is **global** and seen from inside the function.
* `a` is **local** to the function. trying to print outside function throws error.

Note that arguments are essentially locally defined variables.

```
x = 10

def scope_func1(a):
    out = x + a
    return out
```

The following works because `x` is global and functions can access globals so long as they don't reassign the variable name.

```
y = scope_func1(6)
print(y)
```

The following fails because `a` local and not visible outside the function.

```
print(a)
```

60.5 Example 2

x defined both outside and inside function, and used inside the function

This function reassigns `x`, so it becomes local.

Note that a variable becomes local once it is used in an assignment statement within a function, or if it the name of an argument.

```python
x = 10 # Global

def scope_func2(a):

    x = 20 # Local
    print('x from inside:', x)

    out = x + a

    return out
```

```python
y = scope_func2(6)
print(y)
print('x from outside:', x)
```

60.6 Example 2a

Here we define `x` as an argument.

This has the same effect as defining it in the body of the function.

```python
x = 10 # Global

def scope_func2a(a, x=20): # Argment variables are local

    print('x from inside:', x)

    out = x + a

    return out
```

```python
print('x from outside before function:', x)
y = scope_func2a(6)
print(y)
print('x from outside before function:', x)
```

As an argument, though, it can be assigned the value of the global.

Nevertheless, only the value is being passed between the global and local versions of `x`.

```
x = 10 # Global

def scope_func2a(a, x=20):

    print('x from inside before incremenet:', x)
    x += 10
    print('x from inside after incremenet:', x)

    out = x + a

    return out
```

```
print('x from outside before function:', x)
y = scope_func2a(6, x)
print(y)
print('x from outside after function:', x)
```

60.7 Example 3

x defined both outside and inside function, and used inside the function in both global and local modes

This one is interesting. It fails, but it's not clear why at first.

```
x = 10

def scope_func3(a):
    print('x from fcn, before local definition:', x) # Global use of x
    x = 20 # Local use of x
    print('x from fcn, after local definition:', x)
    out = x + a
    return out
```

```
print('x from outside before local definition:', x)
scope_func3(6)
print('x from outside after local definition:', x)
```

The error can be fixed by referencing x as `global` inside function.

```
x = 10

def scope_func4(a):
    global x

    print('x from inside, before local definition:', x)
    x = 20
    print('x from inside, after local definition:', x)

    out = x + a
    return out
```

```
print('x from outside, before local definition:', x)
y = scope_func4(6)
print(y)
print('x from outside, after local definition:', x)
```

Note that the two instances of the variable `z` coexist in the same script because of the rules of scoping.

60.8 Local / global conflicts

What will calling `guess()` do?

Hint: "If you assign a name in any way within a def, it will become a local to that function by default." (Lutz)

```
x = 10

def guess():
    x += 10
    print(x)
```

```
guess()
```

Consider the following expression, which is the same as the unary operation inside of the function `guess()`.

```
x = x + 10
```

The `x` on the left is `local`, since it is being *defined* inside the function.

However, the x on the right is assumed to already be defined, and so is global.

In effect, Python is presented with a contradiction and so throws an error.

We will see that R does not do this; it just goes with the global.

60.9 Nonlocal

If a variable is assigned in an enclosing def, it is nonlocal to nested functions.

The nonlocal keyword is similar to global, except that it refers to the scope of the enclosing function, not the script that contains the funtions.

```
x = 10 # Global
def func1(): # Enclosing function
    x = 20 # Local to function; "Nonlocal" to nested function
    def func1a():
        x = 30 # Local to nested function
        print(x)
    func1a()
    print(x)
```

```
print(x)
func1()
print(x)
```

```
x = 10
def func2():
    x = 20
    def func2a():
        nonlocal x
        x = 30
        print(x)
    func2a()
    print(x)
```

```
print(x) # 1
func2()
print(x) # 4
```

```
x = 10
def func3():
    x = 20
    def func3a():
        global x
        x = 30
        print(x)
    func3a()
    print(x)
```

```
print(x)
func3()
print(x)
```

60.10 Namespaces

Definitions of scope make reference to **namespaces**. Scope and namespaces are closely intertwined concepts. Sometimes it is assumed that the reader knows what this means.

If you've never heard of namespaces, or are unsure of what they are, here's a brief explanation.

A namespace is a system that allows for **a unique name** to associated with each and **every object** in a Python program. * Remember that **an object can be anything** in Python, not just variable, e.g. a function or a class. * Python maintains namespaces internally as **dictionaries**.

A good **analogy** to a namespace system is the **file system** on a computer. You can have files of the same name so long as they are in different folders. The complete name of the file is actually the filename and the names of its parent folders, i.e. the path to the file in the file system.

Another anology is in **human names** – personal names and family names, i.e. first and last names in European countries. These in turn might be contained by larger social groupings.

Finally, another analogy is **home addresses** – house numbers and street names can be reused based on their "path" in the tree of geographic entities that include cities, states, nations, etc.

Similarly, Python understands what exact method or variable one is trying to point to in the code, depending upon the namespace.

Note that in each of these cases, **the data structure is a directed acyclic graph (DAG)**, which is universal structure for organizing unique names.

60.11 Some Visualizations

The same object name can be present in multiple namespaces as isolation between the same name is maintained by their namespace.

Source: “Namespaces and Scope in Python” (GFG).

61 NB: Functions Calling Functions

Purpose: * Illustrate concept of function design * Demonstrate how functions can break down a process into simple components * Demonstrate how component functions build on each other * Introduce idea of functional groups * Motivate use of classes (to be introduced later)

61.1 Basic Insight

Functions contain any code, so they can contain functions. * Functions can call other functions * Functions can define new functions

We create functions that call functions in order to break a complex process into components. * Some functions focus on simple component processes * Other functions combine these into higher order processes * Some functions may be focused on computation, while others may be focused on interacting with users or data sources * We can think of this a division of labor, or "separation of concerns," among functions

When you create groups of functions, they often form natural groups that associated with a common process or task. * These function groups often share variables in addition to calling each ohter

Let's look at some examples to illustrate these points.

61.2 Example 1: Converting Temperatures

Here are three functions that work together to make a temperature converter.

Notice how the last function integrates the first two.

```
def f_to_c(temp):
    """
    Converts F to C and returns a rounded result.
    Expects an integer and returns an integer.
    """
    return round((temp - 32) * (5/9))
```

```
def c_to_f(temp):
    """
    Converts C to F and returns a rounded result.
    Expects an integer and returns an integer.
    """
    return round(temp * (9/5) + 32)

def convert(temp, scale):
    """
    Combines conversion functions into a two-way converter.
    Expects a souce temp (int) and a target scale ('f' or 'c').
    """
    if scale.lower() == "c":
        return f_to_c(temp)  # function call to f_to_c
    else:
        return c_to_f(temp)  # function call to c_to_f
```

Now, here is function that combines the above functions into a **user-facing interface** to the other functions.

```
##| tags: []
def convert_app():
    """
    Provides a user-interface to the the conversion functions.
    """

    # Get user input
    temp = int(input("Enter a temperature: "))
    scale = input("Enter the scale to convert to: (c or f) ")[0].lower()

    # Infer source scale, to be used in the output message
    if scale == 'c':
        current_scale = 'f'
    else:
        current_scale = 'c'

    # Do the conversion
    converted = convert(temp, scale)

    # Print results for user
    print(f"{temp}{current_scale.upper()} is equal to {converted}{scale.upper()}.")
```

```
convert_app()
```

61.2.1 A More Pythonic Solution

We replace if/then statements with dictionary logic.

```
## Put your logic in the data structure
converters = {
    'c': lambda t: (t - 32) * (5/9),
    'f': lambda t: t * (9/5) + 32
}
```

```
def convert_app2():

    # Input from user
    source_temp  = int(input("Enter a temperature: "))
    target_scale = input("Enter the scale to convert to: (c or f) ")

    # Internal computations
    target_temp  = converters[target_scale](source_temp)
    # source_scale = list(set(converters.keys()) - set(target_scale))[0]
    source_scale = (set(converters.keys()) - set(target_scale)).pop()

    # Output to user
    print(source_temp, source_scale, "converted becomes:" , round(target_temp), target_sca
```

```
convert_app2()
```

61.3 Example 2: Counting Vowels

```
## Predicate functions - often used as helper functions that return True or False

def is_vowel(l):
    if l == "a" or l == "e" or l == "i" or l == "o" or l == "u":
        return True  # if the letter is a vowel, return True
    else:
        return False # else, return False
```

```
def num_vowels(my_string):
    my_string = my_string.lower()
    count = 0
    for i in range(len(my_string)): # for each character
        if is_vowel(my_string[i]):  # call function above
            count += 1              # increment count if true
    return count

def vcounter():
    my_str = input("Enter a string: ")
    vcount = num_vowels(my_str)
    print(f"There are {vcount} vowels in the string.")
```

```
vcounter()
```

61.3.1 A More Pythonic Solution

We can use a lambda function with a comprehension to replace the fisrt two functions above.

```
vowel_count = lambda x: len([char for char in x.lower() if char in "aeiou"])
```

```
test_str = "Whatever it is, it is what it is."
```

```
vowel_count(test_str)
```

61.4 Example 3: Calculating Tax

We write two related functions: * One to compute the **tax** based on a **gross pay** and a **tax rate**. * One to compute the **net pay** using the previous function.

In addition, we want to write some functions that use these functions to interact with a user. * One to get the input value of the gross pay and print the tax. * One to print the net pay based on the previous function.

Note the division of labor, or “separation of concerns”, in these functions: * Some do calculative work * Some do interactive work

To compute tax, we have these data:

```
gross_pay    tax_rate
---------------------
0   - 240    0%
241 - 480    15%
481 - more   28%
```

This time, we want to create a group of functions that expect some global variables to exist and use these instead of return statements.

In the code below, we globalize any variables that are assigned in our functions. This allows them to be shared by all the other functions.

Note that this is effective when our global environment – the containing script – contains only these functions. Later in this course, we will look at mechanisms to segment our code in this way.

```
def compute_tax():
    """
    Computes tax rate and applies to gross pay to get tax.
    Expects gross_apy to be defined globally.
    Adds tax_rate and tax to globals for use by other functins.
    """

    global tax_rate, tax

    # Get rate by lower bound
    if gross_pay > 480:
        tax_rate = .28
    elif gross_pay > 240:
        tax_rate = .15
    else:
        tax_rate = 0

    tax = gross_pay * tax_rate

def compute_net_pay():
    """
    Computes net pay based on globals produced by compute_tax().
    Expects gross_pay and tax to be defined globally.
    Adds net_pay to to globals.
    """

    global net_pay

    net_pay = gross_pay - tax

def get_tax():
    """
    Computes and prints tax based on user input.
    Essentially a wrapper around compute_tax().
    Adds gross_pay to globals.
    """

    global gross_pay

    gross_pay = int(input("Enter your gross pay in dollars: "))
    compute_tax()

    print(f"Based on a tax rate of {round(tax_rate * 100)}%, the tax you owe on ${gross_pa

def get_net_pay():
    """
    Computes and prints net pay based on globals.
    """

    compute_net_pay()

    print(f"Your take home (net) pay is ${round(net_pay)}.")

def do_all():
    "Runs both user-facing functions."
    get_tax()
    get_net_pay()
```

```
get_tax()
```

```
get_net_pay()
```

```
do_all()
```

61.5 Concluding Observations

- Notice how each example has functions that build on each other.
- These functions share both data and a general goal.
- The fact that data and functions go together is the motivation for creating classes.