Introduction to Python

August, 2024

Table of Contents

# About this Course

## 0.1 Curriculum

The course covers fundamentals of Python, a high-level programming language, and use it to wrangle data for analysis and visualization.

## 0.2 Target Audience

The course is intended for researchers who want to learn coding for the first time with a data science application via the Python language. This course is also appropriate for folks who have explored data science or programming on their own and want to focus on some fundamentals.

## 0.3 Learning Objectives

**Analyze** Tidy datasets in the Python programming language via data subsetting, joining, and transformations.

**Evaluate** summary statistics and data visualization to understand scientific questions.

**Describe** how the Python programming environment interpret complex expressions made out of functions, operations, and data structures, in a step-by-step way.

**Apply** problem solving strategies to debug broken code.

## 0.4 Offerings

This course is taught on a regular basis at [Fred Hutch Cancer Center](https://www.fredhutch.org/) through the [Data Science Lab](https://hutchdatascience.org/). Announcements of course offering can be found [here](https://hutchdatascience.org/training/).

# 1 Intro to Computing

Welcome to Introduction to Python! Each week, we cover a chapter, which consists of a lesson and exercise. In our first week together, we will look at big conceptual themes in programming, see how code is run, and learn some basic grammar structures of programming.

## 1.1 Goals of the course

In the next 6 weeks, we will explore:

* Fundamental concepts in high-level programming languages (Python, R, Julia, etc.) that is transferable: *How do programs run, and how do we solve problems using functions and data structures?*
* Beginning of data science fundamentals: *How do you translate your scientific question to a data wrangling problem and answer it?*
* 
* Data science workflow. Image source: [R for Data Science](https://r4ds.hadley.nz/whole-game).
* Find a nice balance between the two throughout the course: we will try to reproduce a figure from a scientific publication using new data.

## 1.2 What is a computer program?

* A sequence of instructions to manipulate data for the computer to execute.
* A series of translations: English <-> Programming Code for Interpreter <-> Machine Code for Central Processing Unit (CPU)

We will focus on English <-> Programming Code for Python Interpreter in this class.

More importantly: **How we organize ideas <-> Instructing a computer to do something**.

## 1.3 A programming language has following elements:

* Grammar structure to construct expressions; combining expressions to create more complex expressions
* Encapsulate complex expressions via **functions** to create modular and reusable tasks
* Encapsulate complex data via **data structures** to allow efficient manipulation of data

## 1.4 Google Colab Setup

Google Colab is a Integrated Development Environment (IDE) on a web browser. Think about it as Microsoft Word to a plain text editor. It provides extra bells and whistles to using Python that is easier for the user.

Let’s open up the KRAS analysis in Google Colab. If you are taking this course while it is in session, the project name is probably named “KRAS Demo” in your Google Classroom workspace. If you are taking this course on your own time, open up…

Today, we will pay close attention to:

* Python Console (Execution): Open it via View -> Executed code history. You give it one line of Python code, and the console executes that single line of code; you give it a single piece of instruction, and it executes it for you.
* Notebook: in the central panel of the website, you will see Python code interspersed with word document text. This is called a Python Notebook (other similar services include Jupyter Notebook, iPython Notebook), which has chunks of plain text *and* Python code, and it helps us understand better the code we are writing.
* Variable Environment: Open it by clicking on the “{x}” button on the left-hand panel. Often, your code will store information in the Variable Environment, so that information can be reused. For instance, we often load in data and store it in the Variable Environment, and use it throughout rest of your Python code.

The first thing we will do is see the different ways we can run Python code. You can do the following:

1. Type something into the Python Console (Execution) and type enter, such as 2+2. The Python Console will run it and give you an output.
2. Look through the Python Notebook, and when you see a chunk of Python Code, click the arrow button. It will copy the Python code chunk to the Python Console and run all of it. You will likely see variables created in the Variables panel as you load in and manipulate data.
3. Run every single Python code chunk via Runtime -> Run all.

Remember that the *order* that you run your code matters in programming. Your final product would be the result of Option 3, in which you run every Python code chunk from start to finish. However, sometimes it is nice to try out smaller parts of your code via Options 1 or 2. But you will be at risk of running your code out of order!

To create your own content in the notebook, click on a section you want to insert content, and then click on “+ Code” or “+ Text” to add Python code or text, respectively.

Python Notebook is great for data science work, because:

* It encourages reproducible data analysis, when you run your analysis from start to finish.
* It encourages excellent documentation, as you can have code, output from code, and prose combined together.
* It is flexible to use other programming languages, such as R.

Now, we will get to the basics of programming grammar.

## 1.5 Grammar Structure 1: Evaluation of Expressions

* **Expressions** are be built out of **operations** or **functions**.
* Functions and operations take in **data types**, do something with them, and return another data type.
* We can combine multiple expressions together to form more complex expressions: an expression can have other expressions nested inside it.

For instance, consider the following expressions entered to the Python Console:

18 + 21

## 39

max(18, 21)

## 21

max(18 + 21, 65)

## 65

18 + (21 + 65)

## 104

len("ATCG")

## 4

Here, our input **data types** to the operation are **integer** in lines 1-4 and our input data type to the function is **string** in line 5. We will go over common data types shortly.

Operations are just functions in hiding. We could have written:

from operator import add  
  
add(18, 21)

## 39

add(18, add(21, 65))

## 104

Remember that the Python language is supposed to help us understand what we are writing in code easily, lending to *readable* code. Therefore, it is sometimes useful to come up with operations that is easier to read. (Because the add() function isn’t typically used, it is not automatically available, so we used the import statement to load it in.)

### 1.5.1 Data types

Here are some common data types we will be using in this course.

| Data type name | **Data type shorthand** | **Examples** |
| --- | --- | --- |
| Integer | int | 2, 4 |
| Float | float | 3.5, -34.1009 |
| String | str | “hello”, “234-234-8594” |
| Boolean | bool | True, False |

A nice way to summarize this first grammar structure is using the function machine schema, way back from algebra class:



Function machine from algebra class.

Here are some aspects of this schema to pay attention to:

* A programmer should not need to know how the function is implemented in order to use it - this emphasizes abstraction and modular thinking, a foundation in any programming language.
* A function can have different kinds of inputs and outputs - it doesn’t need to be numbers. In the len() function, the input is a String, and the output is an Integer. We will see increasingly complex functions with all sorts of different inputs and outputs.

## 1.6 Grammar Structure 2: Storing data types in the Variable Environment

To build up a computer program, we need to store our returned data type from our expression somewhere for downstream use. We can assign a variable to it as follows:

x = 18 + 21

If you enter this in the Console, you will see that in the Variable Environment, the variable x has a value of 39.

### 1.6.1 Execution rule for variable assignment

Evaluate the expression to the right of =.

Bind variable to the left of = to the resulting value.

The variable is stored in the Variable Environment.

The Variable Environment is where all the variables are stored, and can be used for an expression anytime once it is defined. Only one unique variable name can be defined.

The variable is stored in the working memory of your computer, Random Access Memory (RAM). This is temporary memory storage on the computer that can be accessed quickly. Typically a personal computer has 8, 16, 32 Gigabytes of RAM. When we work with large datasets, if you assign a variable to a data type larger than the available RAM, it will not work. More on this later.

Look, now x can be reused downstream:

x - 2

## 37

y = x \* 2

It is quite common for programmers to not know what data type a variable is while they are coding. To learn about the data type of a variable, use the type() function on any variable in Python:

type(y)

## <class 'int'>

We should give useful variable names so that we know what to expect! Consider num\_sales instead of y.

## 1.7 Grammar Structure 3: Evaluation of Functions

Let’s look at functions a little bit more formally: A function has a **function name**, **arguments**, and **returns** a data type.

### 1.7.1 Execution rule for functions:

Evaluate the function by its arguments, and if the arguments are functions or contains operations, evaluate those functions or operations first.

The output of functions is called the **returned value**.

Often, we will use multiple functions, in a nested way, or use parenthesis to change the order of operation. Being able to read nested operations, nested functions, and parenthesis is very important. Think about what the Python is going to do step-by–step in the line of code below:

(len("hello") + 4) \* 2

## 18

If we don’t know how to use a function, such as pow() we can ask for help:

?pow  
  
pow(base, exp, mod=None)  
Equivalent to base\*\*exp with 2 arguments or base\*\*exp % mod with 3 arguments  
   
Some types, such as ints, are able to use a more efficient algorithm when  
invoked using the three argument form.

This shows the function takes in three input arguments: base, exp, and mod=None. When an argument has an assigned value of mod=None, that means the input argument already has a value, and you don’t need to specify anything, unless you want to.

The following ways are equivalent ways of using the pow() function:

pow(2, 3)

## 8

pow(base=2, exp=3)

## 8

pow(exp=3, base=2)

## 8

but this will give you something different:

pow(3, 2)

## 9

And there is an operational equivalent:

2 \*\* 3

## 8

## 1.8 Tips on writing your first code

Computer = powerful + stupid

Even the smallest spelling and formatting changes will cause unexpected output and errors!

* Write incrementally, test often
* Check your assumptions, especially using new functions, operations, and new data types.
* Live environments are great for testing, but not great for reproducibility.
* Ask for help!

To get more familiar with the errors Python gives you, take a look at this [summary of Python error messages](https://betterstack.com/community/guides/scaling-python/python-errors/).

# About the Authors

These credits are based on our [course contributors table guidelines](https://www.ottrproject.org/more_features.html#giving-credits-to-contributors).

| Credits | Names |
| --- | --- |
| **Pedagogy** |  |
| Lead Content Instructor(s) | [FirstName LastName](link%20to%20personal%20website) |
| Lecturer(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | Delivered the course in some way - video or audio |
| Content Author(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | If any other authors besides lead instructor |
| Content Contributor(s) (include section name/link in parentheses) - make new line if more than one section involved | Wrote less than a chapter |
| Content Editor(s)/Reviewer(s) | Checked your content |
| Content Director(s) | Helped guide the content direction |
| Content Consultants (include chapter name/link in parentheses or word “General”) - make new line if more than one chapter involved | Gave high level advice on content |
| Acknowledgments | Gave small assistance to content but not to the level of consulting |
| **Production** |  |
| Content Publisher(s) | Helped with publishing platform |
| Content Publishing Reviewer(s) | Reviewed overall content and aesthetics on publishing platform |
| **Technical** |  |
| Course Publishing Engineer(s) | Helped with the code for the technical aspects related to the specific course generation |
| Template Publishing Engineers | [Candace Savonen](https://www.cansavvy.com/), [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/) |
| Publishing Maintenance Engineer | [Candace Savonen](https://www.cansavvy.com/) |
| Technical Publishing Stylists | [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/), [Candace Savonen](https://www.cansavvy.com/) |
| Package Developers ([ottrpal](https://github.com/jhudsl/ottrpal)) [Candace Savonen](https://www.cansavvy.com/), [John Muschelli](https://johnmuschelli.com/), [Carrie Wright](https://carriewright11.github.io/) |  |
| **Art and Design** |  |
| Illustrator(s) | Created graphics for the course |
| Figure Artist(s) | Created figures/plots for course |
| Videographer(s) | Filmed videos |
| Videography Editor(s) | Edited film |
| Audiographer(s) | Recorded audio |
| Audiography Editor(s) | Edited audio recordings |
| **Funding** |  |
| Funder(s) | Institution/individual who funded course including grant number |
| Funding Staff | Staff members who help with funding |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value  
## version R version 4.3.2 (2023-10-31)  
## os Ubuntu 22.04.4 LTS  
## system x86\_64, linux-gnu  
## ui X11  
## language (EN)  
## collate en\_US.UTF-8  
## ctype en\_US.UTF-8  
## tz Etc/UTC  
## date 2024-08-07  
## pandoc 3.1.1 @ /usr/local/bin/ (via rmarkdown)  
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date (UTC) lib source  
## bookdown 0.39.1 2024-06-11 [1] Github (rstudio/bookdown@f244cf1)  
## cachem 1.0.8 2023-05-01 [1] RSPM (R 4.3.0)  
## cli 3.6.2 2023-12-11 [1] RSPM (R 4.3.0)  
## devtools 2.4.5 2022-10-11 [1] RSPM (R 4.3.0)  
## digest 0.6.34 2024-01-11 [1] RSPM (R 4.3.0)  
## ellipsis 0.3.2 2021-04-29 [1] RSPM (R 4.3.0)  
## evaluate 0.23 2023-11-01 [1] RSPM (R 4.3.0)  
## fastmap 1.1.1 2023-02-24 [1] RSPM (R 4.3.0)  
## fs 1.6.3 2023-07-20 [1] RSPM (R 4.3.0)  
## glue 1.7.0 2024-01-09 [1] RSPM (R 4.3.0)  
## htmltools 0.5.7 2023-11-03 [1] RSPM (R 4.3.0)  
## htmlwidgets 1.6.4 2023-12-06 [1] RSPM (R 4.3.0)  
## httpuv 1.6.14 2024-01-26 [1] RSPM (R 4.3.0)  
## knitr 1.47.3 2024-06-11 [1] Github (yihui/knitr@e1edd34)  
## later 1.3.2 2023-12-06 [1] RSPM (R 4.3.0)  
## lifecycle 1.0.4 2023-11-07 [1] RSPM (R 4.3.0)  
## magrittr 2.0.3 2022-03-30 [1] RSPM (R 4.3.0)  
## memoise 2.0.1 2021-11-26 [1] RSPM (R 4.3.0)  
## mime 0.12 2021-09-28 [1] RSPM (R 4.3.0)  
## miniUI 0.1.1.1 2018-05-18 [1] RSPM (R 4.3.0)  
## pkgbuild 1.4.3 2023-12-10 [1] RSPM (R 4.3.0)  
## pkgload 1.3.4 2024-01-16 [1] RSPM (R 4.3.0)  
## profvis 0.3.8 2023-05-02 [1] RSPM (R 4.3.0)  
## promises 1.2.1 2023-08-10 [1] RSPM (R 4.3.0)  
## purrr 1.0.2 2023-08-10 [1] RSPM (R 4.3.0)  
## R6 2.5.1 2021-08-19 [1] RSPM (R 4.3.0)  
## Rcpp 1.0.12 2024-01-09 [1] RSPM (R 4.3.0)  
## remotes 2.4.2.1 2023-07-18 [1] RSPM (R 4.3.0)  
## rlang 1.1.4 2024-06-04 [1] CRAN (R 4.3.2)  
## rmarkdown 2.27.1 2024-06-11 [1] Github (rstudio/rmarkdown@e1c93a9)  
## sessioninfo 1.2.2 2021-12-06 [1] RSPM (R 4.3.0)  
## shiny 1.8.0 2023-11-17 [1] RSPM (R 4.3.0)  
## stringi 1.8.3 2023-12-11 [1] RSPM (R 4.3.0)  
## stringr 1.5.1 2023-11-14 [1] RSPM (R 4.3.0)  
## urlchecker 1.0.1 2021-11-30 [1] RSPM (R 4.3.0)  
## usethis 2.2.3 2024-02-19 [1] RSPM (R 4.3.0)  
## vctrs 0.6.5 2023-12-01 [1] RSPM (R 4.3.0)  
## xfun 0.44.4 2024-06-11 [1] Github (yihui/xfun@9da62cc)  
## xtable 1.8-4 2019-04-21 [1] RSPM (R 4.3.0)  
## yaml 2.3.8 2023-12-11 [1] RSPM (R 4.3.0)  
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library  
##   
## ──────────────────────────────────────────────────────────────────────────────

# 2 References