

# **PyNotes in Agriscience**

Andres Patrignani

# Table of contents

<b>1 Preface</b>	<b>13</b>
1.1 Feedback . . . . .	14
1.2 Support . . . . .	15
1.3 Acknowledgments . . . . .	15
1.4 License . . . . .	15
1.5 References . . . . .	15
<b>I GETTING STARTED</b>	<b>16</b>
<b>2 Inspiration</b>	<b>17</b>
2.1 Great quotes . . . . .	17
<b>3 Reproducible Research</b>	<b>18</b>
3.1 Jupyter Notebooks . . . . .	19
3.2 References and recommended reading . . . . .	19
3.3 Reproducible research questions . . . . .	19
<b>4 Coding Guidelines</b>	<b>21</b>
4.0.1 Define the goal of your code . . . . .	21
4.0.2 Break down the problem (Sketch + pseudocode) . . . . .	21
4.0.3 Adopt version control . . . . .	22
4.0.4 Draft the code . . . . .	22
4.0.5 Error debugging . . . . .	23
4.0.6 Code review . . . . .	23
4.0.7 Refactor code . . . . .	23
4.1 References . . . . .	24
<b>5 Installing packages</b>	<b>25</b>
5.1 Anaconda package . . . . .	25
5.1.1 Step 1: Download the Anaconda installer . . . . .	25
5.1.2 Step 2: Install Anaconda . . . . .	25
5.1.3 Step 3: Open the Anaconda Navigator . . . . .	25
5.2 Git . . . . .	26
5.3 Github . . . . .	26
5.4 Datasets . . . . .	27

<b>6 Useful Terminal Commands</b>	<b>28</b>
6.1 General commands (Mac terminal or Git Bash) . . . . .	28
6.2 Useful shortcuts . . . . .	28
<b>7 Useful Git commands</b>	<b>29</b>
<b>8 First Github repository</b>	<b>31</b>
<b>9 Markdown basics</b>	<b>34</b>
9.1 Comments . . . . .	34
9.2 Line breaks . . . . .	35
9.3 Headers . . . . .	35
9.4 Sub-title header . . . . .	35
9.4.1 Sub-sub-title header . . . . .	35
9.5 Emphasis . . . . .	35
9.6 Highlighting . . . . .	36
9.7 Monospace font . . . . .	36
9.8 Inline equations . . . . .	36
9.9 Block equations . . . . .	36
9.10 Block quotes . . . . .	37
9.11 Bullet lists . . . . .	37
9.12 Numbered Lists . . . . .	37
9.13 In-line links . . . . .	37
9.14 Referenced links . . . . .	38
9.15 Figures . . . . .	38
9.16 Horizontal lines . . . . .	38
9.17 Code . . . . .	38
9.18 Tables . . . . .	39
<b>10 LaTeX equations</b>	<b>41</b>
10.1 Examples . . . . .	41
10.1.1 Reference Evapotranspiration Equation . . . . .	41
10.1.2 Psychrometric constant . . . . .	42
10.1.3 Wind speed at 2 meters above the soil surface . . . . .	42
10.1.4 Mean saturation vapor pressure . . . . .	43
10.1.5 Slope of vapor pressure . . . . .	43
10.1.6 Actual vapor pressure . . . . .	43
10.1.7 Extraterrestrial solar radiation . . . . .	44
<b>11 Python libraries</b>	<b>45</b>
11.1 Biopython . . . . .	45
11.2 EarthPy . . . . .	45
11.3 MetPy . . . . .	45

11.4	pysheds	45
11.5	whitebox	46
11.6	pastas	46
11.7	Rasterio	46
11.8	xarray	46
11.9	sklearn	46
<b>II</b>	<b>BASIC CONCEPTS</b>	<b>47</b>
<b>12</b>	<b>Basic operations</b>	<b>48</b>
12.1	Hello World	48
12.2	Comments	49
12.3	Arithmetic operations	49
12.4	Simple computations	50
12.4.1	Example 1: Unit conversions	51
12.4.2	Example 2: Compute the hypotenuse	52
12.4.3	Example 3: Calculate soil pH	53
12.4.4	Example 4: Calculate bulk density	53
12.4.5	Example 5: Compute grain yield	54
12.5	Python is synchronous	54
12.6	Practice	55
<b>13</b>	<b>Import modules</b>	<b>56</b>
13.1	Importing module syntax	56
<b>14</b>	<b>Data types</b>	<b>60</b>
14.1	Integers	60
14.2	Floating point	60
14.3	Strings	61
14.4	Booleans	65
14.5	Conversion between data types	66
<b>15</b>	<b>Data structures</b>	<b>69</b>
15.1	Lists	69
15.2	Tuples	71
15.3	Dictionaries	72
15.4	Sets	73
15.5	Practice	75
<b>16</b>	<b>Working with dates and times</b>	<b>76</b>
16.0.1	List of shorthands to represent date and time components	76
16.1	Basic <code>datetime</code> Syntax	76
16.2	Working with <code>timedelta</code>	77

16.3 Using Pandas for datetime operations . . . . .	77
<b>17 Indexing and slicing . . . . .</b>	<b>79</b>
17.1 Syntax for indexing and slicing one-dimensional arrays . . . . .	79
17.2 Syntax for indexing and slicing two-dimensional arrays . . . . .	80
17.3 References . . . . .	82
<b>18 If statements . . . . .</b>	<b>83</b>
18.1 Syntax . . . . .	83
18.2 Example 1: Saline, sodic, and saline-sodic soils . . . . .	83
18.3 Example 2: Climate classification based on aridity index . . . . .	84
18.4 Comparative anatomy of If statements . . . . .	85
18.5 References . . . . .	88
<b>19 Functions . . . . .</b>	<b>89</b>
19.1 Syntax . . . . .	89
19.2 Example function: Compute vapor pressure deficit . . . . .	89
19.2.1 Define function . . . . .	90
19.2.2 Description of function components . . . . .	92
19.2.3 Call function . . . . .	93
19.2.4 Evaluate function performance . . . . .	95
19.2.5 Access function help (the docstring) . . . . .	95
19.3 Function variable scope . . . . .	96
19.4 Python handy functions: map, filter, and reduce . . . . .	98
19.4.1 map . . . . .	98
19.4.2 filter . . . . .	99
19.4.3 reduce . . . . .	100
19.5 Comparative anatomy of functions . . . . .	100
19.6 Practice . . . . .	102
<b>20 Lambda functions . . . . .</b>	<b>103</b>
20.1 Syntax . . . . .	103
20.2 Example 1: Convert degrees Fahrenheit to Celsius . . . . .	103
20.3 Breakdown of Lambda function components . . . . .	104
20.4 Example 2: Estimate atmospheric pressure from altitude . . . . .	105
20.5 Example 3: Volume of tree trunk . . . . .	106
20.6 Example 4: Calculate the sodium adsorption ratio (SAR) . . . . .	107
20.7 Example 5: Compute soil porosity . . . . .	107
20.8 Practice . . . . .	108
<b>21 Inputs . . . . .</b>	<b>109</b>
21.1 Practice . . . . .	110

<b>22 For loop</b>	<b>111</b>
22.1 Syntax . . . . .	111
22.2 Example 1: Basic For loop . . . . .	111
22.3 Example 2: For loop using the <code>enumerate</code> function . . . . .	112
22.4 Example 3: Combine <code>for</code> loop with <code>if</code> statement . . . . .	112
22.5 Example 4: For loop using a dictionary . . . . .	113
22.6 Example 5: Nested for loops . . . . .	114
22.7 Example 6: For loop using <code>break</code> and <code>continue</code> . . . . .	115
22.8 Compatitive anatomy of <code>for</code> loops . . . . .	116
<b>23 While loop</b>	<b>118</b>
23.1 Syntax . . . . .	118
23.2 Example 1: A trivial loop . . . . .	118
23.3 Example 2: Guess the soil taxonomic order . . . . .	119
23.3.1 Explanation . . . . .	120
23.4 References . . . . .	121
<b>24 Objects and classes</b>	<b>122</b>
24.1 Syntax . . . . .	122
24.2 Properties and methods example . . . . .	123
24.3 Class example: Laboratory sample . . . . .	124
24.4 Practice . . . . .	127
<b>25 Error handling</b>	<b>128</b>
25.1 Syntax . . . . .	128
25.2 Example: Classification of soil acidity-alkalinity . . . . .	129
25.3 Explanation . . . . .	130
<b>26 Numpy module</b>	<b>131</b>
26.1 Element-wise computations using Numpy arrays . . . . .	131
26.1.1 Product of a regular list by a scalar . . . . .	131
26.1.2 Product of two regular lists with the same shape . . . . .	132
26.1.3 Product of a Numpy array by a scalar . . . . .	132
26.1.4 Product of two Numpy arrays with the same shape . . . . .	133
26.1.5 Other operations with Numpy arrays . . . . .	133
26.2 Example 1: Compute soil water storage for a single field . . . . .	133
26.3 Example 2: Compute soil water storage for multiple fields . . . . .	134
26.3.1 Example 3: Determine the CEC of a soil . . . . .	135
26.3.2 Create arrays with specific data types . . . . .	136
26.3.3 Operations with two-dimensional arrays . . . . .	137
26.3.4 Reshape arrays . . . . .	138
26.4 Numpy boolean operations . . . . .	138
26.5 Flattening . . . . .	139

26.6 Use the Numpy random module to create a random image . . . . .	139
26.7 Numpy handy functions . . . . .	141
26.8 Create a noisy wave . . . . .	143
26.8.1 Descriptive stats . . . . .	144
26.9 Reference . . . . .	145
<b>27 Pandas module</b>	<b>146</b>
27.1 Create DataFrame from existing variable . . . . .	146
27.2 Basic methods and properties . . . . .	147
27.3 Convert strings to datetime . . . . .	148
27.4 Extract information from the timestamp . . . . .	149
27.5 Missing values . . . . .	150
27.6 Quick statistics . . . . .	152
27.7 Indexing and slicing . . . . .	154
27.7.1 Using index operator [] . . . . .	154
27.8 Select rows . . . . .	154
27.9 Select columns . . . . .	154
27.10 Slicing rows and columns . . . . .	155
27.10.1 Using iloc method . . . . .	155
27.10.2 Using loc method . . . . .	156
27.11 Filter data using boolean indexing . . . . .	157
27.12 Pandas custom date range . . . . .	159
27.13 Select range of dates with boolean indexing . . . . .	159
27.14 Add and remove columns . . . . .	160
27.15 Reset DataFrame index . . . . .	161
27.16 Merge two dataframes . . . . .	162
27.17 Operations with real dataset . . . . .	164
27.17.1 Match specific stations . . . . .	165
<b>28 Plotting</b>	<b>168</b>
28.1 Dataset . . . . .	168
28.2 Matplotlib module . . . . .	169
28.2.1 Components of Matplotlib figures . . . . .	169
28.2.2 Matplotlib syntax . . . . .	171
28.2.3 Line plot . . . . .	174
28.2.4 Scatter plot . . . . .	176
28.2.5 Histogram . . . . .	178
28.2.6 Subplots . . . . .	180
28.2.7 Secondary Y axis plots . . . . .	183
28.3 Bokeh module . . . . .	185
28.4 Interactive line plot . . . . .	186
28.5 Seaborn module . . . . .	187
28.5.1 Line plot . . . . .	188

28.5.2 Correlation matrix . . . . .	188
28.5.3 Scatterplot matrix . . . . .	189
28.5.4 Heatmap . . . . .	190
28.5.5 Boxplot . . . . .	191
<b>29 Widgets</b>	<b>193</b>
29.1 Example 1: Convert bushels to metric tons . . . . .	193
29.2 Example 2: Runoff-Precipitation . . . . .	194
<b>30 SQLite Database</b>	<b>196</b>
30.1 Additional software . . . . .	196
30.2 Key commands . . . . .	196
30.3 Data types . . . . .	197
30.4 Set up a simple database . . . . .	197
30.4.1 Add Data . . . . .	198
30.4.2 Add with multiple entries . . . . .	198
30.4.3 Query data . . . . .	199
30.4.4 Modify data . . . . .	200
30.4.5 Remove data . . . . .	201
30.4.6 Add new column/header . . . . .	202
30.4.7 Closing the connection . . . . .	202
30.5 Use Pandas to set up database . . . . .	202
30.5.1 Connect, access all data, and close database . . . . .	205
<b>III EXERCISES</b>	<b>207</b>
<b>31 Meteogram</b>	<b>208</b>
31.1 Estimate some useful metrics . . . . .	210
31.2 Meteogram . . . . .	211
<b>32 Group with least variance</b>	<b>216</b>
32.1 Practice . . . . .	219
<b>33 Random Plot Generator</b>	<b>220</b>
33.1 Complete Randomized Design . . . . .	221
33.2 Complete Randomized Block . . . . .	221
33.3 References . . . . .	222
<b>34 Particle Random Walk</b>	<b>223</b>
34.1 Solution without for loop . . . . .	224
34.2 Shortest solution . . . . .	225

<b>35 Runoff</b>	<b>226</b>
35.1 Practice . . . . .	229
35.2 References . . . . .	229
<b>36 Mixing problems</b>	<b>230</b>
36.1 Example 1: Tank level and salt concentration . . . . .	230
36.1.1 Step 1: Find time it takes to fill the tank . . . . .	231
36.1.2 Step 2: Add the calculation of the amount of salt . . . . .	231
36.1.3 Example 2: Excess of herbicide problem . . . . .	235
36.2 References . . . . .	236
<b>37 Mass-volume relationships</b>	<b>237</b>
37.1 Problem 1 . . . . .	237
37.2 Problem 2 . . . . .	239
37.3 Problem 3 . . . . .	240
<b>38 Soil textural classes</b>	<b>241</b>
38.1 References . . . . .	247
<b>39 Distribution daily precipitation</b>	<b>248</b>
39.1 Read and prepare dataset for analysis . . . . .	248
39.2 Find value and date of largest daily rainfall event on record . . . . .	249
39.3 Probability density function of precipitation amount . . . . .	249
39.4 Cumulative density function . . . . .	251
39.5 References . . . . .	253
<b>40 High-resolution rainfall events</b>	<b>255</b>
40.1 References . . . . .	262
<b>41 First and last frost</b>	<b>263</b>
41.1 Find date of last frost . . . . .	264
41.2 Find date of first frost . . . . .	264
41.3 Find median date for first and last frost . . . . .	265
41.4 Compute frost-free period . . . . .	266
41.5 Probability density functions . . . . .	266
<b>42 Air Temperature</b>	<b>272</b>
42.1 Compute annual average air temperature . . . . .	274
42.2 Compute daily thermal amplitude . . . . .	274
42.3 Examine residuals . . . . .	278
42.4 Compute Tmax and Tmin for each DOY . . . . .	279
42.5 Compute annual thermal amplitude . . . . .	281
42.6 Build model for air temperature . . . . .	281
42.7 Practice . . . . .	283

<b>43 Potential Evapotranspiration</b>	<b>284</b>
43.1 Dalton model . . . . .	286
43.2 Romanenko model . . . . .	286
43.3 Penman model . . . . .	287
43.4 Jensen model . . . . .	287
43.5 Hargreaves model . . . . .	287
43.5.1 Priestley-Taylor model . . . . .	288
43.6 Penman-Monteith model . . . . .	289
43.7 Data . . . . .	290
43.8 Practice . . . . .	291
43.9 References . . . . .	291
<b>44 Growing degree days</b>	<b>292</b>
44.1 Example using non-vectorized functions . . . . .	295
44.2 Example using vectorized functions . . . . .	298
44.3 Practice . . . . .	301
44.4 References . . . . .	302
<b>45 Central dogma</b>	<b>303</b>
45.1 Transcription . . . . .	303
45.2 Translation . . . . .	304
<b>46 Frontier production functions</b>	<b>307</b>
46.1 Curve interpretation . . . . .	313
46.2 Additional summary metrics . . . . .	314
46.3 Quantile regression . . . . .	315
46.4 Observations . . . . .	318
46.5 References . . . . .	318
<b>47 Soil Water Storage</b>	<b>319</b>
47.1 Trapezoidal integration . . . . .	320
47.2 Contour plot . . . . .	323
47.3 References . . . . .	323
<b>48 Plant Available Water</b>	<b>324</b>
48.1 Integral energy . . . . .	324
48.2 Soil water storage . . . . .	327
48.3 Energy . . . . .	328
48.4 References . . . . .	328
<b>49 Photoperiod</b>	<b>329</b>
49.1 References . . . . .	332

<b>50 Solar Radiation</b>	<b>333</b>
50.1 Inspect timeseries . . . . .	334
50.2 Clear sky irradiance (empirical) . . . . .	334
50.3 Clear sky irradiance (from latitude) . . . . .	335
50.4 Actual solar irradiance (from air temperature) . . . . .	337
50.5 Clear sky solar radiation for each DOY . . . . .	337
<b>51 Potential Evapotranspiration</b>	<b>339</b>
51.1 Dalton model . . . . .	341
51.2 Romanenko model . . . . .	341
51.3 Penman model . . . . .	342
51.4 Jensen model . . . . .	342
51.5 Hargreaves model . . . . .	342
51.5.1 Priestley-Taylor model . . . . .	343
51.6 Penman-Monteith model . . . . .	344
51.7 Data . . . . .	345
51.8 Practice . . . . .	346
51.9 References . . . . .	346
<b>52 Soil Temperature Model</b>	<b>347</b>
52.1 Model . . . . .	347
52.2 Assumptions . . . . .	348
52.3 Model inputs . . . . .	348
52.4 Define model . . . . .	349
52.5 Soil temperature for a specific depth as a function of time . . . . .	349
52.6 Soil temperature for a specific day of the year as a function of depth . . . . .	350
52.7 Soil temperature as a function of both DOY and depth . . . . .	350
52.8 Interactive plots . . . . .	352
52.9 References . . . . .	354
<b>53 Soil Water Retention Curve</b>	<b>355</b>
53.1 Define soil water retention model . . . . .	356
53.2 Fit model . . . . .	357
53.3 References . . . . .	359
<b>54 Proctor test</b>	<b>360</b>
54.1 References . . . . .	364
<b>55 Counting seeds</b>	<b>365</b>
<b>56 Canopy cover</b>	<b>373</b>
56.1 Read and process a single image . . . . .	373
56.2 References . . . . .	379

<b>57 Cleaning yield monitor data</b>	<b>380</b>
57.1 Dataset description . . . . .	380
57.2 File formats . . . . .	381
57.3 Filtering rules with clear physical meaning . . . . .	384
57.4 Moving filters . . . . .	388
57.4.1 Function to compute Euclidean distance . . . . .	388
57.4.2 Moving median filter . . . . .	388
57.4.3 Moving filter to detect outliers . . . . .	390
57.5 Observations . . . . .	392
57.6 References . . . . .	392

# 1 Preface

Welcome to a journey at the intersection of programming, data science, and agronomy. This book presents hands-on coding exercises designed to address common tasks in crop and soil sciences.

Coding is an essential component of modern scientific research that enables more creative solutions, in-depth analyses, and ensures reproducibility of findings. This material is part of an introductory graduate level course offered to students in plant and soil sciences with little or no coding experience. Anyone with sufficient motivation, discipline, and interest in learning how to code and adopt reproducible research practices should (hopefully) find the content of this notes useful. The material is aimed at students that are transitioning from spreadsheets analysis to a programming environment and that first need to learn basic building blocks before tackling more complicated challenges. With most datasets in a tabular format, the material is easily accessible and inspectable using common spreadsheet software.

I selected Python because of its accessibility (it's free), straightforward syntax, multi-purpose applications (data analysis, desktop applications, websites, games), widespread adoption in the scientific community, and a rich ecosystem of tools for reproducible research that makes the transition into the coding world a lot easier to beginners. The code presented here is complemented by live coding lectures and might not always reflect the most efficient or 'pythonic' methods. The goal is to present clear and explicit code to gradually familiarize students with the Python syntax, documentation resources, and improve the process of breaking down problems into a sequence of smaller logical steps to ultimately reach more advanced and elegant coding. This book strives for a balanced approach, blending task complexity with a judicious use of libraries. While libraries enhance reproducibility and benefit from extensive testing of the community, an overreliance on them can hinder beginners from truly grasping the underlying concepts and logic of programming.

The motivation for these notes stem from the need to increase coding literacy in students pursuing a career in agricultural sciences. With the rise of sensor technology and the expanding volume of data in agronomic decision-making, scientific programming has become indispensable for agriscientists analyzing, interpreting, and communicating data and research findings. This material addresses three key gaps:

1. A scarcity of online resources with real agricultural datasets. This series uses data from peer-reviewed studies and research projects, offering practical and applicable examples that bridge theory and practice.

2. Existing coding resources often target either a general audience or advanced students in computer science, leaving a void for agronomy students and early career scientists in agricultural sciences that are new to coding.
3. Aiming to provide concise, interactive, and well-documented Jupyter notebooks, this material is readily accessible via platforms like [Github](#) and [Binder](#).

During my own journey in graduate school, coding was a powerful tool for enhancing logical thinking, deconstructing complex problems into manageable steps, and sharpening my focus on details that initially seemed inconsequential. Code brought to life abstract concepts and equations that appeared in manuscripts and books, making intricate processes more tangible and comprehensible.

As a faculty member, coding has reshaped the way I interact with students. Code reveals the student's reasoning process and allows me to connect with the student logical (or sometimes illogical) thought process. This way I can somewhat get into the student's head and correct misconceptions. Collaborative coding has become one of the most fulfilling aspects of my academic career.

The goal of this book is to make you a **competent** (based on the [Dreyfus scale](#)) python programmer, meaning that you will be able to automate simple tasks, analyze datasets, and create reproducible and logical code that supports scientific claims in the domain of agricultural sciences. Students who successfully complete the material will be able to:

- construct effective, well-documented, and error-free scripts and functions.
- apply high-level programming to generate publication-quality figures and optimize simple models.
- find information independently for self-teaching and problem solving.
- learn good programming habits and basic reproducible research practices by following short exercises using real data.

I truly hope you find both learning and enjoyment in these pages. Happy coding!

Andres Patrignani Associate Professor in Soil Water Processes Department of Agronomy  
Kansas State University

## 1.1 Feedback

If you encounter any errors or have suggestions for improvement, please, share your insights with me. For bug reports, code suggestions, or requests for new topics, please create an issue in the Github repository. This platform is ideal for collaborative discussion and tracking the progress of your suggestions. You can also contact me directly at [andrespatrignani@ksu.edu](mailto:andrespatrignani@ksu.edu).

## **1.2 Support**

The content of this website is partially supported by the Kansas State University [Open/Alternative Textbook Initiative](#)

## **1.3 Acknowledgments**

This book was enriched by the invaluable insights and encouragement from numerous faculty members and students. Their inspiration and constructive feedback have been pivotal in shaping the content you see today. I am deeply grateful for their contributions and the collaborative spirit that has permeated this endeavor over the past ten years.

## **1.4 License**

All the code in these Jupyter notebooks has been written entirely by the author unless noted otherwise. The entire material is available for free under the Creative Commons Attribution-NonCommercial-ShareAlike ([CC BY-NC-SA](#)) license

## **1.5 References**

Dreyfus, S.E., 2004. The five-stage model of adult skill acquisition. Bulletin of science, technology & society, 24(3), pp.177-181. <https://doi.org/10.1177/0270467604264992>

# **Part I**

# **GETTING STARTED**

## 2 Inspiration

A short video created by [code.org](#) to increase awareness about the importance of coding literacy. The footage includes the humble beginnings of successful coders and entrepreneurs that revolutionized the tech industry through code.

### 2.1 Great quotes

These are great quotes obtained from the video and the official code.org website:

“I think that great programming is not all that dissimilar to great art. Once you start thinking in concepts of programming it makes you a better person...as does learning a foreign language, as does learning math, as does learning how to read.”  
—*Jack Dorsey. Creator, Twitter. Founder and CEO, Square*

“Software touches all of these different things you use, and tech companies are revolutionizing all different areas of the world...from how we shop to how farming works, all these things that aren’t technical are being turned upside down by software. So being able to play in that universe really makes a difference.” —*Drew Houston. Founder & CEO, Dropbox*

“To prepare humanity for the next 100 years, we need more of our children to learn computer programming skills, regardless of their future profession. Along with reading and writing, the ability to program is going to define what an educated person is.” —*Salman Khan. Founder, Khan Academy*

“Learning to speak the language of information gives you the power to transform the world.” —*Peter Denning. Association of Computing Machinery, former President*

“Learning to write programs stretches your mind, and helps you think better, creates a way of thinking about things that I think is helpful in all domains.” —*Bill Gates. Chairman, Microsoft*

“The programmers of tomorrow are the wizards of the future. You’re going to look like you have magic powers compared to everybody else” —*Gabe Newell. Founder and President, Valve*

## 3 Reproducible Research

The core principle of reproducible research is to supplement published studies not only with datasets from field or lab observations, model outputs, or instrument readings, but also with the code used in data analysis. This ensures that others can validate and replicate the findings of the study.

One of the primary benefits of utilizing a high-level programming language is its capacity to handle data in a cross-platform format, like .txt or .csv. This allows for seamless integration of various tasks, including data wrangling, numerical and statistical analyses, geospatial analyses, and the creation of publication-quality visuals within a single coding environment. Essentially, the aim is to centralize all stages of data analysis on one platform. By using code to document our thought processes and methodologies, we can reduce or even eliminate the reliance on multiple specialized software programs, thereby enhancing the transparency of our research.

### Note

Have you ever counted the number of mouse clicks and keyboard strokes needed to redo an entire data analysis for a manuscript when juggling between several software applications?

Here are my top three reproducible research practices for research:

1. **Avoid manipulation of raw data:** This includes adding/removing/editing raw data in files created by dataloggers, data retrieved from online sources, or field/lab spredsheets. If you are copy-pasting data, you are probably doing it wrong. The idea is to handle tasks like replacing missing values, removing outliers, and skipping unnecesary data with a programming language. An exception may be if the file was corrupted (say by a malfunctioning datalogger) that prevents us from reading the file.
2. **Document and structure your code so that is human-readable:** This step requires adding comments to lines of code, making use of white space or cells to break down the code into smaller sections, add equations with reference to papers, books, or manuals, and document the code with additional explanations, equations, and figures using markdown.
3. **Provide public access to the data and associated code:**

- by using a dedicated version-control platforms for creating and sharing code like Github and Gitlab. Even tools like Dropbox and OneDrive are a step forward for making your research findings reproducible if you don't feel comfortable with other platforms,
- uploading files to a general-purpose open-access repository like Zenodo, which also generates a digital object identifier (DOI) that can be used in future citations,
- by hosting the datasets and code in your own research website.

### **3.1 Jupyter Notebooks**

A Jupyter notebook is a web-based environment for interactive computing. Jupyter notebooks seamlessly aggregate executable code, comments, equations, images, references, and paths or URL links to specific datasets within a single platform. In a Jupyter notebook, the code is neatly compartmentalized into cells, offering an organized and intuitive structure for coding. These cells are the cornerstone of a Jupyter Notebook's functionality, allowing for the execution of individual code segments (activated by pressing `ctrl + enter`) independently. This feature enables coders to test and validate each block of code separately, ensuring its functionality and correctness before proceeding to subsequent sections. This modular approach to code execution not only enhances the debugging process but also improves the overall development workflow.

### **3.2 References and recommended reading**

Guo, P., 2013. Helping scientists, engineers to work up to 100 times faster. [Link](#)

Shen, H., 2014. Interactive notebooks: Sharing the code. *Nature*, 515(7525), pp.151-152. [Link](#)

Sandve, G.K., Nekrutenko, A., Taylor, J. and Hovig, E., 2013. Ten simple rules for reproducible computational research. *PLoS computational biology*, 9(10). [Link](#)

Skaggs, T.H., Young, M.H. and Vrugt, J.A., 2015. Reproducible research in vadose zone sciences. *Vadose Zone Journal*, 14(10). [Link](#)

### **3.3 Reproducible research questions**

Based on the reading of Guo 2013 and Skaggs et al., 2015, answer the following questions:

Q1. List and briefly explain all the softwares that you used in the past 3 years for data analysis as part of your research.

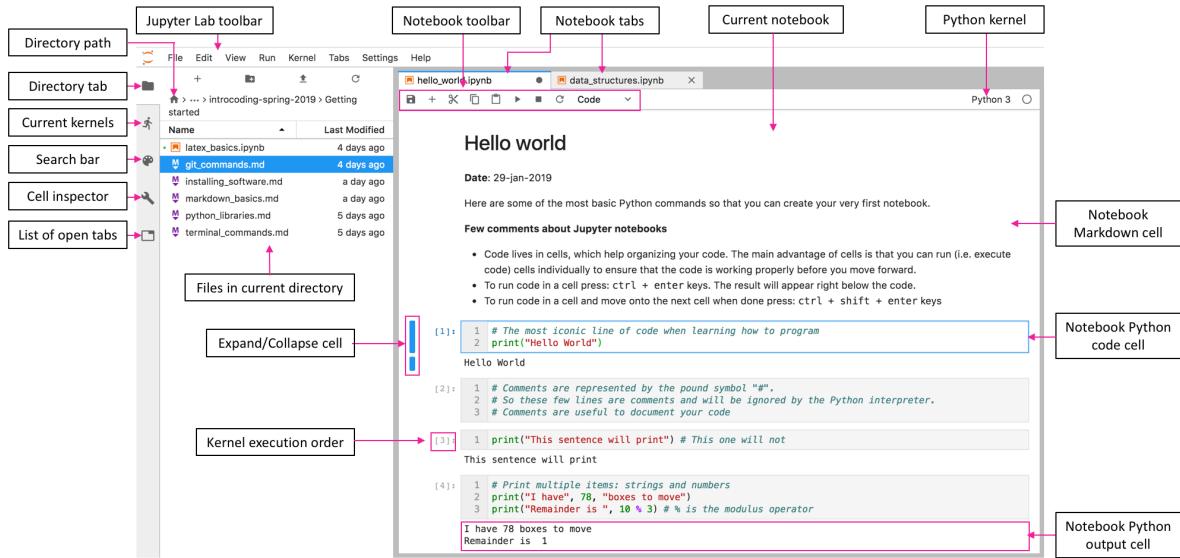


Figure 3.1: Graphical user interface of Jupyter Lab notebooks.

Q2. Briefly define what is reproducible research?

Q3. Name 3 reasons why you need to learn coding as a scientist or engineer.

Q4. How do you feel about sharing your data and code with the rest of the scientific community when publishing an article? Do you have any concerns?

# 4 Coding Guidelines

The steps below are aimed at new coders to help them identify the purpose of the code, stimulate the process of breaking down problems into smaller pieces, and logically organize these smaller components or steps. Writing clean and human-readable code requires consistency and the adoption of good programming practices and habits.

## 4.0.1 Define the goal of your code

Briefly describe the purpose of the code. What do you want the code to do? Here it is important to be as specific as possible.

## 4.0.2 Break down the problem (Sketch + pseudocode)

To decompose the problem into smaller steps it is a good idea to sketch the components and flow of your tentative code. At this point you may not know exactly all the steps and complications of your code. Iterative sketching and drafting will clarify your ideas. If your code relies on multiple equations, add them to the sketch so that you can build the logical flow of your code.

For instance, describe whether inputs will be retrieved from a remote server, the file system, or from the user through an interactive interface. Identify potential functions that may emerge from this work and that can be re-used throughout the code.

### 💡 Tip

In this step is important that you consider starting from scratch all over again if necessary. Sometimes we get entangled with our own ideas and a fresh start can be the best solution. I recommend using a canvas like a whiteboard or a piece of paper that you can easily erase.

#### **4.0.3 Adopt version control**

To keep track of changes and backup your code it is an imperative practice in modern science to adopt version control tools (e.g. git), whether you are part of a team or if you are working alone. Although maybe not ideal, even tools like Dropbox and OneDrive will be a step forward and may allow inexperienced coders that feel intimidated by more advanced tools to remain engaged.

#### **4.0.4 Draft the code**

This is the part where we write and test the code for the first time. Below is a list of good coding habits that you may want to consider during this process:

- At the top of your code, include your name, date, a brief description of the purpose of the code, and, if necessary, provide the reference to a manuscript, book, or blog.
- Code one or a few lines at the time.
- Before running any code, set an expectation for the output. Whether you are running a simple `print` statement or several lines of code, ensuring that you are obtaining the desired outcome will prevent that errors accumulate and propagate in your code.
- Use cells in Jupyter notebooks to break down the problem into smaller pieces. Write and test the code for each cell before proceeding to the next part of the problem.
- Add one to three sentences at the beginning of any function or script describing the purpose of the code (in functions this is called the *docstring*).
- If you follow the steps or equations detailed in a peer-reviewed manuscript or book, add a comment next to each line referencing the equation number or page number.
- When defining inputs, outputs, and the intermediate variables add a comment with the units.
- Use descriptive variable names. Err on the verbose side instead of assigning cryptic variable names. Exceptions may apply when coding equations from a specific source.
- Write descriptive variable names using underscores (e.g. `air_temperature`)
- Function names should be all lower case with no underscores (e.g. `thermaltime()`).
- Use spaces to improve readability, but avoid extraneous whitespace inside parentheses, brackets, and braces. For instance: `c = (a+b) * (a-b)` is better than `c = (a + b) * (a - b)`.
- If a block of code is long and complicated, consider splitting the code into smaller blocks. If this is not possible, use white space and meaningful comments to explain the steps of the code.

### Note

I highly encourage you to check the [Python style guide](#) for detailed guidelines and examples:

#### **4.0.5 Error debugging**

Encountering errors and making mistakes is a natural part of the coding process. Rather than expecting perfection at every step, embrace these moments as critical learning opportunities. I recommend writing a line of code, then testing it to ensure it behaves as expected before moving on. Always have a clear expectation for each line's output - this approach not only helps in identifying errors early but also deepens your understanding of how your code functions.

### Tip

A frequent pitfall among new programmers is assuming a line of code is correct without thoroughly testing or questioning the logical validity of its output. Always approach code with a clear expectation of the outcome, enabling you to effectively assess the validity of the results.

One of the more frustrating aspects of coding is how small syntax errors, like a missing parenthesis, an extra comma, or a misspelled function name, can stop your code from running entirely. When you're new to coding, these details can be easily overlooked. Almost every coder can recall a time they spent an entire afternoon troubleshooting a simple typo. Therefore, it's crucial to meticulously examine your code to ensure everything is correct and functioning as intended.

#### **4.0.6 Code review**

Ensure that others read your code. Don't take criticism personally. If something is not clear it might be due to bad syntax, improper logic, or lack of documentation. If you are the reviewer, be respectful, honest, and provide constructive feedback.

#### **4.0.7 Refactor code**

This step is about polishing the code by implementing one or more of the following: adding comments, renaming variables with more meaningful names, merging cells with related code for a more compact code, re-writing lines of code in a more time-efficient way, replacing code with more modular functions, removing unnecessary code, replacing loops by element-wise operations, pre-allocating memory, etc.

## 4.1 References

Van Rossum, G., Warsaw, B. and Coghlan, N., 2001. PEP 8: style guide for Python code. Python. org, 1565. The official Python style guide (PEP 8) can be found [here](#).

# 5 Installing packages

To follow the material in this book you will need to install the Anaconda package, Git, and create an account in Github. We will write code using the web browser, so I recommend having Google Chrome installed.

## 5.1 Anaconda package

We will use the Anaconda environment, which is a set of curated python packages commonly used in science and engineering. The Anaconda environment is available for free by Continuum Analytics.

### 5.1.1 Step 1: Download the Anaconda installer

- Download the [Anaconda package](#) for your platform (Windows, Mac, Linux)

### 5.1.2 Step 2: Install Anaconda

- Double click on the installer and follow the steps. When asked, I highly suggest installing VS Code, which is a powerful editor with autocomplition, debugging capabilities, etc.
- In case you are having trouble, visit the Anaconda “Frequently Asked Questions” for some tips on how to troubleshoot most common issues: <https://docs.anaconda.com/anaconda/user-guide/faq/>

### 5.1.3 Step 3: Open the Anaconda Navigator

- In Windows go to the start up menu in the bottom left corner of the screen and then click on the Anaconda Navigator.
- In Macs go to Applications and double click on the Anaconda Navigator. Alternatively you can use the search bar (press `Command + Space bar` and search for terminal).
- **JupyterLab and Jupyter Notebook:** We will write most of our code using notebooks, which are ideal for reproducible research.

- **VS Code:** A powerful and modern code editor. You can download it and code here if you want.
- **Spyder:** An integrated development environment for scientific coding in Python. It features a graphical user interface similar to that of Matlab.

**i** Note

If you open a notebook and run the `pip list` command, you can print all the installed packages in your Anaconda environment.

## 5.2 Git

### What is Git?

Git is a distributed version control system that enables multiple users to track and manage changes to code and documents

### How do I get started with Git?

If you have a Mac, you most likely already have Git installed. If you have a Windows machine or need to install it for your Mac, follow these steps:

1. Go to: <https://git-scm.com>
2. Select Windows/MacOS
3. Follow the installer and use default installation settings
4. We will most use the command window (called Git Bash), but we need it in order to work with Github.

## 5.3 Github

### What is Github?

- GitHub is a web platform that hosts Git repositories, offering tools for collaboration, code review, and project management. In addition to Github, there are other similar platforms such as Bitbucket and GitLab.

### How do I get started with Github?

1. Create a Github account at: <https://github.com/>
2. Create a repository. Make sure to add a **README** file.
3. Go to your computer and open the terminal
4. Navigate to a directory where you want to place the repository

5. Clone the Github repository using: git clone <link>

These are just a few short instructions. Check out the detailed and more extensive tutorial to get started.

## 5.4 Datasets

Most examples and exercises in the book use real datasets, which can be found in the `/datasets` directory of the [Github repository](#). You can download the entire repository, a specific file, or simply read the file using the “Raw” URL link. For example, to read the daily weather dataset for the Kings Creek watershed named `kings_creek_2022_2023_daily.csv` you can run the following command:

```
pd.read_csv(https://raw.githubusercontent.com/andres-patrignani/harvestingdatawithpython/main/datasets/kings\_creek\_2022\_2023\_daily.csv)
```

# 6 Useful Terminal Commands

While some tools like Github now have some graphical user interfaces, terminal commands can be quite handy for cloning repositories, committing changes, pushing updates, and even installing new python modules. Something you almost always need to do in the terminal is to navigate directories. Here are some of the most typical commands and shortcuts to get you started.

## 6.1 General commands (Mac terminal or Git Bash)

`cd <foldername>` change directory; navigate into a new folder (assuming the folder exists in the directory).

`cd ..` To navigate out of the current directory

`ls` To list the content of the folder

`pwd` full path to current directory

See image below where I ran these commands:

## 6.2 Useful shortcuts

Use `Tab` key to autocomplete directory and file names

Use `arrow-up` to access the last command

In Macs use `cmd + v` to paste text into the terminal

In Windows machines use `shift + insert` to paste text into Git bash

## 7 Useful Git commands

Some of the most widely used git terminal commands. You can run the commands below anywhere, you don't need to be inside any specific directory or repository. Some of these commands will change information in Git's configuration file, so that you don't have to re-type your username and email everytime you make a commit.

`git config --global user.name "john-doe"` In this case my username is: john-doe. The dash in the middle is part of the username.

`git config --global user.email johndoe@abc.org` In this case see that I did not include the email between quotation marks as I did with the username.

`git config --global core.editor "nano"` changes the default editor from "vim" to "nano". In my opinion **nano** is a bit more friendly for beginners. Exit nano by pressing **ctrl + X**

`.gitignore` contains file and folder names that you don't want to keep track of version control. In other words, they will not sync with Github. If you added a rule in the `.gitignore` file after the file or folder has been added to your Github, you will need to erase the cache of the repository and then add the files again, so that changes take effect. You can do this following these commands: `git rm --cached -r .` and then `git add .`

`git clone <repository link>`: Clone repository into your local computer or a remote server. You only clone your repository once. If you work on a server or supercomputer, cloning a repository from a cloud-based platform like Github is much easier than transferring files using the terminal or copy pasting files using a graphical user interface like FileZilla.

`git status`: This command provides the state of the current repository in the local computer (or server) relative to Github's remote repository.

`git add .`: Adds and removes **all** new file additions/deletions from repository. Remember, when you add or remove a new file to a folder in your local computer, it does not get automatically added to your repository. You have to execute the command for this to happen.

`git add <filename>` In case you want to add a single file to your repository. `<filename>` could be `mytextfield.txt`

`git commit -a -m "<write here a short descriptive message>"`: Send changes to remote repository. **Messages are mandatory** and should be short and descriptive. Don't accumulate too many changes before committing, otherwise your message/comment will not be meaningful to all the changes you made.

`git push origin master` Upload changes to master branch in the Github repository. To see changes make sure you refresh the Github webpage.

`git pull origin master`: Downloads updates in the master branch.

`git checkout -- ..`: Disregard changes.

`git checkout <branch name>`: Changes scope to any branch, including the master branch. This command assumes that you have a branch.

## 8 First Github repository

This example will guide you through the step by step creation of a Github repository. Before we start, it will be helpful if we define some of the jargon involved in the Git commands:

**clone:** Means to download or copy an entire repository. **push:** Send updates/changes to Github repository. You need to clone the repository first. **pull:** Get updates from Github repository

1. Install the latest version of Git in your computer.
2. Create a Github account at [github.com](https://github.com) and log in.
3. Create a new repository
  - Typical repository naming convention is: this-my-first-repo
  - Make sure to add a README file
4. Then, copy the download link.
5. In your computer, open the terminal or GitBash and navigate to the directory in which you would like to download (**clone**) your repository.
6. Write: `git clone <link>` to clone the repository to your computer. Replace `<link>` with the link that you copy in the previous step using `cmd + v` in macs and `ctrl + insert` in Windows machines.
7. Now the repository is in your computer (and of course in Github, we did it in step 3).  
**> Important:** To do the following push or pull commands you first need to clone your repository.
8. At this point there is only one file in the repository, the `README.md` file. `md` files are basically text files with some markup formatting. Github will render Markdown text in this file and display it as the landing page of your repository.
9. As an example, let's try to edit the `README` file. This statement is tricky because it is unclear whether I'm asking you to edit the `README` file in your local computer or to do so in Github. Yes, you can edit files in Github (at least text files). Let's edit the `README` file in your computer.
10. Open the `README` file in a Jupyter Lab, Jupyter Notebook, or a regular text editor. Add some text to it. Don't worry about adding Markdown format, just add some content.

11. Save the modifications.
12. Now the README file in your computer has modifications that the README file in Github doesn't. Github does not sync automatically (like Dropbox). So, we need to find a way to send the changes to Github.
13. In your computer, use the terminal to navigate inside your repository.
14. Run the following commands in the terminal. Press the `enter` key after writing each command. Enter Github account credentials if needed:

```
git add .
git commit -a -m "Updated README file"
git push
```

We didn't really add a new file, so the first command is not required here, but it does not hurt to add it (at least in most scenarios when you are getting started).

15. Go to your Github account and refresh the page. Since Github renders the content in the README file as the landing page of your repository, you should be able to see the changes right in front of your eyes.
16. Follow step 13 and 14 to send future updates to your Github repository.
17. If in step 9 you decided to edit the README file directly in Github, then the situation would be the opposite. You would have changes in the README file in your Github account that aren't in your local computer. To get the updates into your local computer, open the terminal and navigate to your repository and type `git pull`. Open the README file with a text editor and you should see the changes you made in the Github website.
18. Sometimes there are files that you just want to have them in your local computer, but don't want to upload to your Github repository. These files may include sensitive data, large files (>100 MB), or temporary files created by applications. So, to prevent Git syncing these files we a list of files that we want Git to ignore. This file is called the `.gitignore` file. To create a `.gitignore` file open a Jupyter Lab and go to: File -> New -> Text File. Click rename, delete the entire filename (make sure you also delete the .md part), and then name it `.gitignore`. The leading period is important. Make sure that there isn't a trailing `.txt` extension. This means that this is hidden file. The Jupyter Lab navigation panel will not display hidden files. Just search on the web how to make hidden files visible in Windows Explorer or Mac Finder. The `.gitignore` file will appear in both your local directory and your Github repository. > Note that if you first add undesirable files to your repository (using `git add`, `commit`, and `push`), adding the `.gitignore` will not delete them from your Github account. If you are in this situation just insert: `git rm --cached -r .` and then `git add .`

**Example .gitignore file** Text between square brackets is a comment and you shouldn't write it into your .gitignore file.

```
.DS_Store  
**/.ipynb_checkpoints/  
/Private notebooks
```

# 9 Markdown basics

[Markdown](#) is a simple and minimalist markup language aimed at reducing distractions caused by the continuous need to format text. Markdown enables writers to focus on writing. Jupyter notebooks can combine code with documentation written in Markdown to provide additional information without cluttering the code with comments. Markdown also enables the use of rich media such as hyperlinks, images, videos, and equations. These features make Jupyter notebooks great for creating both reproducible research articles and teaching content.

Markdown is a markup language created by John Gruber to speed up formatting without having to write HTML (HyperText markup Language, the language we use to write websites). Markdown is eventually translated into HTML code. There is no global standard, and there are slightly different variations of Markdown. The Github-flavored (Kramdown) markdown official documentation can be found here: <https://github.github.com/gfm/>

The list of examples below is only aimed at getting you started. I encourage you to visit the following two links for detailed syntax, cheatsheets, and more comprehensive examples:

- [Excellent syntax guide](#)
- [Adam Pritchard's popular Markdown examples](#)
- [John Gruber's original specifications](#)

## 9.1 Comments

Markdown does not seem to have an official way of adding comments. However, we can fool several Markdown interpreters by preceding text with the following expression [//]:

[//]: This is a comment

Note that this trick might not work in some Markdown editors like Typora, but it does seem to work in Github.

## 9.2 Line breaks

Pressing the `enter` key will not generate empty lines. Because Markdown eventually is converted into HTML, we can use HTML tags to expand the editing and styling possibilities in our document. So, to add a line break, we can use the self-closing line break tag: `<br/>`.

```
some text  
</br>  
more text
```

## 9.3 Headers

Represented by adding 1 to 6 leading `#` signs

```
# Title header  
## Sub-title header  
### Sub-sub-title header
```

## 9.4 Sub-title header

### 9.4.1 Sub-sub-title header

## 9.5 Emphasis

```
*italic text*  
_italic text_  
**bold text**  
__bold text__  
~~striked text~~
```

*italic text*

*italic text*

**bold text**

**bold text**

~~text~~

## 9.6 Highlighting

To calculate the `sin(90)` first import the `math` module`

To calculate the `sin(90)` first import the `math` module`

## 9.7 Monospace font

Indent text using the Tab key to generate a monospace font.

## 9.8 Inline equations

`$y = ax+b$`

$$y = ax + b$$

## 9.9 Block equations

Example equation for calculating actual vapor pressure (Eq. 17, FAO-56):

`$$ea = \frac{eTmin\frac{RHmax}{100}+eTmax\frac{RHmin}{100}}{2}$$`

$$ea = \frac{eTmin \frac{RHmax}{100} + eTmax \frac{RHmin}{100}}{2}$$

`ea` = actual vapor pressure (kPa)

`eTmax` = saturation vapor pressure at temp Tmax (kPa)

`eTmin` = saturation vapor pressure at temp Tmin (kPa)

`RHmax` = maximum relative humidity (%)

`RHmin` = minimum relative humidity (%)

## 9.10 Block quotes

Use the > character to generate block quotes.

```
>"The programmers of tomorrow are the wizards of the future. You're going to look like you have magic powers compared to everybody else."
```

“The programmers of tomorrow are the wizards of the future. You’re going to look like you have magic powers compared to everybody else.” - *Gabe Newell*

## 9.11 Bullet lists

Any of these two alternatives:

- item 1 \* item 1
- item 2 \* item 2
- item 3 \* item 3

will generate something similar to this: - item 1 - item 2 - item 3

## 9.12 Numbered Lists

1. item 1
2. item 2
3. item 3

1. item 1
2. item 2
3. item 3

## 9.13 In-line links

```
[Github-flavored markdown(https://www.wikiwand.com/en/Home\_page)
```

Github-flavored markdown

## 9.14 Referenced links

```
[Try a live Markdown editor in your browser] [1]
```

```
Some text  
Some more text
```

```
[1] https://stackedit.io "Optional title to identify your source"
```

[Try a live Markdown editor in your browser](#)

## 9.15 Figures

Figures can be inserted in Markdown following this syntax:

```
![alt_text](https://path_to_my_image/image.jpg "My image")
```

Because we many times want to deploy our Markdown in Github, then using pure HTML is the best option:

```

```

## 9.16 Horizontal lines

You can use three consecutive dashes, asterisks, or underscores in this fashion:

```
---
```

```
***
```

```
---
```

For instance, typing ---, we obtain the following line:

---

## 9.17 Code

We can write inline or block code. Inline code:

```
`s = "Python inline code syntax highlighting"`

s = "Python inline code syntax highlighting"
```

and block code:

```
```python
# Creating a matrix or 2D array
M = [[1, 4, 5],
      [-5, 8, 9]]
print(M)
```
```

```
# Creating a matrix or 2D array
M = [[1, 4, 5],
      [-5, 8, 9]]
print(M)
```

## 9.18 Tables

Simple tables are easy to write in Markdown. However, adding more than a handful of rows and/or columns can turn out to be a pain. So, if you want to display many lines I suggest using a Markdown table generator. Some Markdown editors have shortcuts and table generators and there are websites exclusively dedicated to generate Markdown tables. Below I show a trivial example:

| Textural class  | Sand (%) | Clay (%) |
|-----------------|----------|----------|
| Silty clay loam | 10       | 35       |
| Sandy loam      | 60       | 15       |
| Clay loam       | 35       | 35       |

The leftmost column is left-aligned :---, the center column is center-aligned :---:, and the rightmost column is right-aligned ---:. The | characters don't need to be aligned in order for the Mardown interpreter to properly render the table, but it certainly helps while constructing the table by hand.

| Textural class  | Sand (%) | Clay (%) |
|-----------------|----------|----------|
| Silty clay loam | 10       | 35       |
| Sandy loam      | 60       | 15       |

| Textural class | Sand (%) | Clay (%) |
|----------------|----------|----------|
| Clay loam      | 35       | 35       |

# 10 LaTeX equations

In the notebooks we use Markdown to write text, but equations are rendered using LaTeX syntax. LaTeX is a typesetting that allows technical writers and scientists to focus on the content without worrying about the format. LaTeX is free under the terms of the LaTeX Project Public License (LPPL).

!>A nice feature of LaTeX is that there is a specific syntax for equations. For instance, you can define inline equations so that this: `$y=mx+b$`, get's converted into this:  $y = mx + b$

It is also possible to write equations in separate lines using `$$y = e^{-x}$$`:

$$y = e^{-x}$$

## 10.1 Examples

Below is a set of equations obtained from the [FAO 56 manual](#) to calculate reference evapotranspiration. Use this equations as templates to learn how to implement your own equations.

### 10.1.1 Reference Evapotranspiration Equation

`$$ETo = \frac{0.408\Delta(Rn-G)+\gamma\frac{900}{T+273}u2(es-ea)}{\Delta+\gamma(1+0.34u2)}$$`

$$ET_o = \frac{0.408\Delta(Rn - G) + \gamma \frac{900}{T+273} u^2 (es - ea)}{\Delta + \gamma(1 + 0.34u^2)}$$

$ET_o$  = reference evapotranspiration (mm/day)

$Rn$  = net radiation at the crop surface (MJ/m<sup>2</sup>/day)

$G$  = soil heat flux density (MJ/m<sup>2</sup>/day)

$T$  = mean daily air temperature at 2 m height

$u^2$  = wind speed at 2 m height (m/s)

$es$  = saturation vapor pressure (kPa)

$ea$  = actual vapor pressure (kPa)

$es - ea$  = saturation vapor pressure deficit (kPa)

$\Delta$  = slope vapor pressure curve (kPa/°C)

$\gamma$  = psychrometric constant (kPa/°C)

### 10.1.2 Psychrometric constant

$$\gamma = \frac{C_p P}{\epsilon \lambda}$$

$\gamma$  = psychrometric constant (kPa/°C)

$\lambda$  = latent heat of vaporization, 2.45 (MJ/kg)

$C_p$  = specific heat at constant pressure (MJ/kg/°C)

$\epsilon$  = ratio of molecular weight of water vapour/dry air = 0.622

$P$  = atmospheric pressure (kPa)

### 10.1.3 Wind speed at 2 meters above the soil surface

$$u2 = uz \frac{4.87}{\ln(67.8z - 5.42)}$$

$$u2 = uz \frac{4.87}{\ln(67.8z - 5.42)}$$

$u2$  = wind speed at 2 m above ground surface (m/s)

$uz$  = measured wind speed at  $z$  m above ground surface (m/s)

$zm$  = height of measurement above ground surface (m)

#### 10.1.4 Mean saturation vapor pressure

$$es = \frac{eT_{max} + eT_{min}}{2}$$

$$es = \frac{eT_{max} + eT_{min}}{2}$$

$es$  = mean saturation vapor pressure (kPa)

$eT_{max}$  = saturation vapor pressure at temp  $T_{max}$  (kPa)

$eT_{min}$  = saturation vapor pressure at temp  $T_{min}$  (kPa)

#### 10.1.5 Slope of vapor pressure

$$\Delta = \frac{4098 \left[ 0.6108 \exp \left( \frac{17.27 T_{mean}}{T_{mean} + 237.3} \right) \right]}{(T_{mean} + 237.3)^2}$$

$\Delta$  = slope of saturation vapor pressure curve at air temp  $T$  (kPa/°C)

$T_{mean}$  = average daily air temperature

#### 10.1.6 Actual vapor pressure

$$ea = \frac{eT_{min} \frac{RH_{max}}{100} + eT_{max} \frac{RH_{min}}{100}}{2}$$

$$ea = \frac{eT_{min} \frac{RH_{max}}{100} + eT_{max} \frac{RH_{min}}{100}}{2}$$

$ea$  = actual vapor pressure (kPa)

$eT_{max}$  = saturation vapor pressure at temp  $T_{max}$  (kPa)

$eT_{min}$  = saturation vapor pressure at temp  $T_{min}$  (kPa)

$RH_{max}$  = maximum relative humidity (%)

$RH_{min}$  = minimum relative humidity (%)

### 10.1.7 Extraterrestrial solar radiation

$$Ra = \frac{24(60)}{\pi} G \int dr [\omega \sin(\phi) \sin(\delta) + \cos(\phi) \cos(\delta) \sin(\omega)]$$

$$Ra = \frac{24(60)}{\pi} G dr [\omega \sin(\phi) \sin(\delta) + \cos(\phi) \cos(\delta) \sin(\omega)]$$

$Ra$  = extraterrestrial radiation (MJ/m<sup>2</sup>/day)

$G$  = solar constant (MJ/m<sup>2</sup>/min)

$dr = 1 + 0.033 \cos(2\pi J/365)$

$J$  = number of the day of the year

$\phi = \pi/180$  decimal degrees (latitude in radians)

$\delta = 0.409 \sin((2\pi J/365) - 1.39)$  Solar decimation (rad)

$\omega = \pi/2 - (\arccos(-\tan(\phi) \tan(\delta)))$  sunset hour angle (radians)

# **11 Python libraries**

Note that module descriptions were obtained from their respective official site.

## **11.1 Biopython**

Mature module for biological computation developed and maintained by a global community. I suggest inspecting the tutorial and the cookbook examples. Great, clean documentation.

Official page: <https://biopython.org/>

## **11.2 EarthPy**

Collection of IPython notebooks with examples of Earth Science. The module was developed and is maintained by Nikolay Koldunov. Examples are available in the “DataproCESSing” tab.

Official page: <http://earthpy.org/>

## **11.3 MetPy**

Open Source project for meteorological data analysis.

Official page: <https://unidata.github.io/MetPy/latest/>

## **11.4 pysheds**

Library for watershed delineation in Python.

Site: <https://github.com/mdbartos/pysheds>

## **11.5 whitebox**

Python package to perform common geographical information systems analysis operations such as cost-distance analysis, distance buffering, and raster reclassification. It also has a GUI interface.

Official site: <https://github.com/giswqs/whitebox>

## **11.6 pastas**

Python package for processing, simulating and analyzing hydrological time series.

Official site: <https://github.com/pastas/pastas>

## **11.7 Rasterio**

Library that allows you to read, organize, and store gridded raster datasets such as satellite imagery and terrain models in GeoTIFF and other formats.

Official site: <https://rasterio.readthedocs.io/en/latest/>

## **11.8 xarray**

Python project for working with labelled multi-dimensional arrays.

Official site: <http://xarray.pydata.org/en/stable/>

## **11.9 sklearn**

Data mining, data analysis, and machine learning library to solve classification, regression, and clustering problems.

Official site: <https://scikit-learn.org/stable/>

## **Part II**

# **BASIC CONCEPTS**

# 12 Basic operations

We start our Python journey with the most traditional step in learning any programming language: writing a “Hello World” program. This simple exercise is a rite of passage for all new programmers. It involves writing a short piece of code that displays the phrase “Hello World” on the screen. While it might seem elementary, this exercise is crucial as it introduces you to the basic structure of a Python program and the process of executing code. It’s your first step into the world of coding, where you begin to understand how a few lines of code can create an output, setting the stage for more complex and exciting projects ahead.

Below we will cover the following concepts:

- Hello World
- Variables
- Comments
- Arithmetic operators
- Simple computations
- Use the `print()` command
- Python help
- Importing modules from the Python Standard library
- Python synchronous interpreter

## 💡 Tip

Press the `Ctrl + Enter` keys to run code in a cell. You can also press the `Ctrl + Shift + Enter` keys to run code in a cell and move onto the next cell.

## 12.1 Hello World

```
# The most iconic line of code when learning how to program
print("Hello World")
```

```
Hello World
```

## 12.2 Comments

In Python, as in any programming language, comments play a crucial role in enhancing the clarity, readability, and maintainability of code. Comments are used to annotate various parts of the code to provide context or explain the purpose of specific blocks or lines. This is especially important in Python, where the emphasis on clean and readable code aligns with the language's philosophy. Comments are a mark of good programming practice and help both the original author and other programmers who may work with the code in the future to quickly understand the intentions, logic, and functionality of the code.

**In Python, a comment is created by placing the hash symbol # (a.k.a. pound sign, number sign, sharp, or octothorpe) before any text or number.** Anything following the # on the same line is ignored by the Python interpreter, allowing programmers to include notes and explanations directly in their code.

```
# This line is a comment and will be ignored by the Python interpreter.  
print("This sentence will print") # This sentence will not print
```

This sentence will print



Tip

Comments can be used to disable code lines without deleting them. This is especially handy when experimenting with different solutions. You can retain alternative code snippets as comments, which will be ignored by the interpreter, but remain available for reference or reactivation at a later time.

## 12.3 Arithmetic operations

In Python, arithmetic operators are fundamental tools used for performing basic mathematical operations. The most commonly used operators include addition (+), subtraction (-), multiplication (\*), and division (/). Apart from these, Python also features the modulus operator (%) which returns the remainder of a division, the exponentiation operator (\*\*) for raising a number to the power of another, and the floor division operator (//) which divides and rounds down to the nearest whole number. These operators are essential building blocks and form the basis of more complex mathematical functionality in the language.

```
print(11 + 2)    # Addition  
print(11 - 2)    # Subtraction  
print(11 * 2)    # Multiplication
```

```
print(11 / 2)    # Division
print(11 // 2)   # Floor division
print(11**2)     # Exponentiation
print(11 % 2)    # Modulus
```

```
13
9
22
5.5
5
121
1
```

### 💡 Tip

In newer versions of Jupyter Lab you can run a single line within a cell by: 1. Selecting the line you want to run 2. Going to the Run menu 3. Selecting **Run Selected Text or Current Line in Console**

### ⚠️ Exponentiation

In Python, the exponentiation operation is performed using a double asterisk (\*\*), which is a departure from some other programming languages, such as Matlab, where a caret (^) is used for this operation.

## 12.4 Simple computations

In programming, a variable is like a storage box where you can keep data that you might want to use later in your code. Think of it as a named cell in an Excel spreadsheet. Just as you might store a number or a piece of text in a spreadsheet cell and refer to it by its cell address (like A1 or B2), in coding, you store data in a variable and refer to it by the name you have given it. Variables are essential because they allow us to handle data dynamically and efficiently in our programs.

To get comfortable with the language syntax, variable definition, data types, and operators let's use Python to solve simple problems that we typically carry on a calculator. In the following examples you learn how to:

- code and solve a simple arithmetic problem
- document a notebook using Markdown syntax
- embed LaTeX equations in the documentation

### 12.4.1 Example 1: Unit conversions

Perhaps one of the most common uses of calculators is to do unit conversions. In this brief example we will use Python to convert from degrees Fahrenheit to degrees Celsius:

$$C = (F - 32) \frac{5}{9}$$



#### Note

Always assign meaningful names to your variables. Variable names must start with a letter or an underscore and cannot contain spaces. Numbers can be included in variable names as long as they are preceded by a letter or underscore.

```
value_in_fahrenheit = 45
value_in_celsius = (value_in_fahrenheit-32) * 5/9

# Print answer
print("The temperature is:", round(value_in_celsius,2), "°C")
```

The temperature is: 7.22 °C



The print and round functions can take multiple arguments as inputs. In this context, the comma is a delimiter between the function inputs. Use the `help()` or `?`  command to access a brief version of the documentation of a function. For instance, both `help(round)` and `round?`  will print the documentation for the `round()` function, showing the first argument is the number we want to round, and the second argument is the number of decimal digits. The official documentation for the `round()` function is available at [this link](#).

```
# Access function help
round?
```

Signature: `round(number, ndigits=None)`

Docstring:

Round a number to a given precision in decimal digits.

The return value is an integer if `ndigits` is omitted or `None`. Otherwise

```
the return value has the same type as the number. ndigits may be negative.  
Type: builtin_function_or_method
```

### 12.4.2 Example 2: Compute the hypotenuse

Given that `a` and `b` are the sides of a right-angled triangle, let's compute the hypotenuse `c`. For instance, if `a = 3.0` and `b = 4.0`, then our code must return a value of `c = 5.0` since:

$$c = \sqrt{a^2 + b^2}$$

```
a = 3.0 # value in cm  
b = 4.0 # value in cm  
hypotenuse = (a**2 + b**2)**(1/2)  
  
# Print answer  
print('The hypotenuse is:', round(hypotenuse, 2), 'cm')
```

```
The hypotenuse is: 5.0 cm
```

**Did you notice that we used `**(1/2)` to represent the square root?** Another alternative is to import the `math` module, which is part of the Python Standard Library, and extends the functionality of our program. To learn more, check the notebook about importing Python modules.

```
import math  
hypotenuse = math.sqrt((a**2 + b**2))  
  
# Print answer  
print('The hypotenuse is:', round(hypotenuse, 2), 'cm')
```

```
The hypotenuse is: 5.0 cm
```

```
# The math module also has a built-in function to compute the Euclidean norm or hypotenuse  
print(math.hypot(a, b))
```

```
5.0
```

### 12.4.3 Example 3: Calculate soil pH

Soil pH represents the  $H^+$  concentration in the soil solution and indicates the level of acidity or alkalinity of the soil. It is defined on a scale of 0 to 14, with pH of 7 representing neutral conditions, values below 7 indicating acidity, and values above 7 indicating alkalinity. Soil pH influences the availability of nutrients to plants and affects soil microbial activity. It is typically measured using a pH meter or indicator paper strips, where soil samples are mixed with a distilled water and then tested.

The formula is:  $pH = -\log_{10}(H^+)$

```
H_concentration = 0.0001
soil_ph = -math.log10(H_concentration)
print(soil_ph)
```

4.0

### 12.4.4 Example 4: Calculate bulk density

The bulk density is crucial soil physical property for understanding soil compaction and soil health. Assume that an undisturbed soil sample was collected using a ring with a diameter of 7.5 cm and a height of 5.0 cm. The soil was then oven-dried to remove all the water. Given a mass of dry soil ( $M_s$ ) of 320 grams (excluding the mass of the ring), calculate the bulk density ( $\rho_b$ ) of the soil using the following formula:  $\rho_b = M_s/V$ . In this case we assume that the ring was full of soil, so the volume of the ring is the volume of the soil under analysis.

```
dry_soil = 320 # grams
ring_diamter = 7.5 # cm
ring_height = 5.0 # cm
ring_volume = math.pi * (ring_diamter/2)**2 * ring_height # cm^3

bulk_density = dry_soil/ring_volume

# Print answer
print('Bulk density =', round(bulk_density, 2), 'g/cm^3')
```

Bulk density = 1.45 g/cm<sup>3</sup>

### ⚠ Tip

In Python 3, [UTF-8](#) is the default character encoding, so Unicode characters can be used anywhere. This is why we can write  $^{\circ}\text{C}$  and  $\text{cm}^3$ . [Here](#) is an extensive list of Unicode characters

#### 12.4.5 Example 5: Compute grain yield

Computing grain yield is probably one of the most common operations in the field of agronomy. In this example we will compute the grain yield for a corn field based on kernels per ear, the number of ears per plant, the number of plants per square meter, and the weight of 1,000 kernels. This example can be easily adapted to estimate the harvestable yield of other crops.

```
# Define variables
kernels_per_ear = 500
ears_per_plant = 1
plants_per_m2 = 8
weight_per_1000_kernels = 285 # in grams

# Calculate total kernels per square meter
kernels_per_m2 = kernels_per_ear * ears_per_plant * plants_per_m2

# Calculate yield in grams per square meter
# We divide by 1000 to compute groups of 1000 kernels
yield_g_per_m2 = (kernels_per_m2 / 1000) * weight_per_1000_kernels

# Convert yield to kilograms per hectare
# Use (1 hectare = 10,000 square meters) and (1 metric ton = 1 Mg = 1,000,000 g)
yield_tons_per_ha = yield_g_per_m2 * 10000 / 10**6

print("Corn grain yield in metric tons per hectare is:", yield_tons_per_ha)
```

Corn grain yield in metric tons per hectare is: 11.4

#### 12.5 Python is synchronous

An important concept demonstrated by the `time` module is Python's synchronous nature. The Python interpreter processes code line by line, only moving to the next line after the current one has completed execution. This sequential execution means that if a line of code requires

significant time to run, the rest of the code must wait its turn. While this can cause delays, it also simplifies coding by allowing for a straightforward, logical sequence in scripting. With the `sleep()` method, we can introduce a time delay to simulate extended computations.

```
print('Executing step 1')
time.sleep(5) # in seconds

print('Executing step 2')
time.sleep(3) # in seconds

print('Executed step 3')

print('See, Python is synchronous!')
```

```
Executing step 1
Executing step 2
Executed step 3
See, Python is synchronous!
```

## 12.6 Practice

Create a notebook that solves the following problems. Make sure to document your notebook using Markdown and LaTeX.

1. Convert 34 acres to hectares. *Answer: 13.75 hectares*
2. Compute the slope (expressed as a percentage) between two points on a terrain that are 150 meters apart and have a difference in elevation of 12 meters. *Answer: 8% slope*
3. Compute the time that it takes for a beam of sunlight to reach our planet. The Earth-Sun distance is 149,597,870 km and the speed of light in vacuum is 300,000 km per second. Express your answer in minutes and seconds. *Answer: 8 minutes and 19 seconds*

### Syntax tip

Code readability is a core priority of the Python language. Since Python 3.6, now we can use underscores to make large numbers more readable. For instance, the distance to the sun could be written in Python as: 149\_597\_870. The Python interpreter will ignore the underscores at the time of performing computations.

# 13 Import modules

Python modules, also known as packages or toolboxes, are collections of pre-written Python code that provide additional functionality. Some of these modules are part of the Python standard library, readily available for use without any additional installation. Other modules can be installed separately as needed, expanding the range of functionalities beyond the standard library. Python is a thriving ecosystem with programmers constantly developing new modules. You can find most packages in the [Python Package Index \(PyPI\)](#) repository of software for the Python language.

## Import modules convention

Modules are imported once, usually at the top of the code or notebook. throughout more advance exercise in this book you will see a code cell near the top of the code solely dedicated to importing modules.

## 13.1 Importing module syntax

There are multiple ways of importing Python modules depending on whether you want to import the entire module, assign a shorter alias, or import a specific sub-module. While the examples below could be applied to the same module, I chose to use typical examples that you will encounter for each option.

**Option 1** Syntax: `import <module>` Example: `import math` Use-case: This is the simplest form of importing and is used when you need to access multiple functions or components from a module. It's commonly used for modules with short names that don't require an alias. When using this form, you call the module's components with the module name followed by a dot, e.g., `math.sqrt()`.

```
import math

a = 3.0 # value in cm
b = 4.0 # value in cm
hypotenuse = math.sqrt((a**2 + b**2))
print('The hypotenuse is:', round(hypotenuse,2), 'cm')
```

```
The hypotenuse is: 5.0 cm
```

Here is another example using this syntax. With the `sys` module we can easily check our python version

```
import sys
print(sys.version) # Useful to check your python version
```

```
3.10.9 (main, Mar 1 2023, 12:33:47) [Clang 14.0.6 ]
```

**Option 2** Syntax: `import <module> as <alias>` Example: `import numpy as np` Use-case: This is used to import the entire module under a shorter, often more convenient alias. It's particularly useful for frequently used modules or those with longer names. The alias acts as a shorthand, making the code more concise and easier to write. For instance, you can use `np.array()` instead of `numpy.array()`.

```
import numpy as np

# Create an Numpy array of sand content for different soils
sand_content = np.array([5, 20, 35, 60, 80]) # Percentage

# Create an Numpy array of porosity values for the same soils above
porosity = np.array([55, 50, 47, 40, 35]) # Percentage

print(sand_content)
print(porosity)
```

```
[ 5 20 35 60 80]
```

```
[55 50 47 40 35]
```

**Option 3** Syntax: `import <module>.<submodule> as <alias>` Example: `import matplotlib.pyplot as plt` Use-case: This approach is used when you only need a specific part or submodule of a larger module. It's efficient as it only loads the required submodule into memory. The alias is used for ease of reference, as in `plt.plot()` instead of using `matplotlib.pyplot.plot()`, which will be much more verbose and will result in cluttered lines of code.

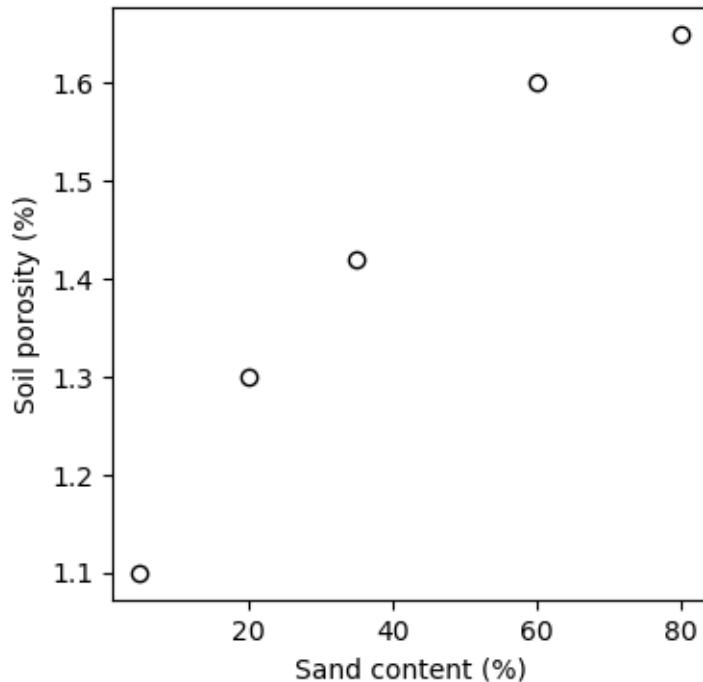
```
import matplotlib.pyplot as plt

plt.figure(figsize=(4,4))
```

```

plt.scatter(sand_content, bulk_density, facecolor='white', edgecolor='black')
plt.xlabel('Sand content (%)')
plt.ylabel('Soil porosity (%)')
plt.show()

```



**Option 4** Syntax: `from <module> import <submodule> as <alias>` Example: `from numpy import random as rnd` Use-case: This method is used when you need only a specific component or function from a module. It's the most specific and memory-efficient way of importing, as only the required component is loaded. This is useful for modules where only one or a few specific functions are needed, and it allows you to use these functions directly without prefixing them with the module name, such as using `rand()` instead of `numpy.random()`. Despite being more memory efficient, this option can lead to conflicts in the variable namespace. This can cause confusion if multiple imported elements share the same name, potentially overwriting each other. Additionally, this approach can obscure the origin of functions or components, reducing code readability.

```

from numpy import random as rnd

# Set seed for reproducibility (without this you will get different array values)
rnd.seed(0)

```

```
# Create a 5 by 5 matrix of random integers between 0 and 9
# The randint function returns
M = rnd.randint(0, 10, [5,5])
print(M)
```

```
[[5 0 3 3 7]
 [9 3 5 2 4]
 [7 6 8 8 1]
 [6 7 7 8 1]
 [5 9 8 9 4]]
```

**i** Note

Use the Python help to access the documentation of the `randint()` method by running the following command: `rnd.randint?` Can you see why we had to use a value of 10 for the second argument of the function?

# 14 Data types

In Python, data types are categories that determine the kind of value an object can have and the operations that can be performed on it. The most basic data types include integers (`int`), floating-point numbers (`float`), strings (`str`), and booleans (`bool`). Each of these types serves a specific purpose: integers for whole numbers, floats for decimal numbers, strings for text, and booleans for true/false values. Understanding these data types is crucial as they form the building blocks of Python coding, influencing how you store, handle, and interact with data in your programs.

## 14.1 Integers

```
plants_per_m2 = 8
print(plants_per_m2)
print(type(plants_per_m2)) # This is class int
```

```
8
<class 'int'>
```

## 14.2 Floating point

```
rainfall_amount = 3.4 # inches
print(rainfall_amount)
print(type(rainfall_amount)) # This is class float (numbers with decimal places)
```

```
3.4
<class 'float'>
```

## 14.3 Strings

```
# Strings are defined using single quotes ...
common_name = 'Winter wheat'

# ... or double quotes
scientific_name = 'Triticum aestivum'

# ... but do not mix them

print(common_name)
print(scientific_name)

print(type(common_name))
print(type(scientific_name))

Winter wheat
Triticum aestivum
<class 'str'>
<class 'str'>

# For longer blocks that span multiple lines we can use triple quotes ('' or "")

soil_definition = """The layer(s) of generally loose mineral and/or organic material
that are affected by physical, chemical, and/or biological processes
at or near the planetary surface and usually hold liquids, gases,
and biota and support plants."""

print(soil_definition)
```

The layer(s) of generally loose mineral and/or organic material  
that are affected by physical, chemical, and/or biological processes  
at or near the planetary surface and usually hold liquids, gases,  
and biota and support plants.

### Note

The multi-line string appears as separate lines due to hidden line breaks at the end of each line, like when we press the Enter key. These breaks are represented by \n,

which are not displayed, but can be used for splitting the long string into individual lines. Try the following line: `soil_definition.splitlines()`, which is equivalent to `soil_definition.split('\n')`

```
# Split a string
filename = 'corn_riley_2024.csv' # Filename with corn yield data for Riley county in 2024

# Split the string and assign the result to different variables
# This only works if the number of variables on the LHS matches the number of outputs
# Try running filename.split('.') on its own to see the resulting list
base_filename, ext_filename = filename.split('.')
print(base_filename)

# Now we can do the same, but splitting at the underscore
crop, county, year = base_filename.split('_')

print(crop)
```

corn\_riley\_2024  
corn

### i Note

The command `base_filename, ext_filename = filename.split('.')` splits the string at the ., and automatically assigns each of the resulting elements ('corn\_riley\_2024' and 'csv') to each variable on the left-hand side. If you only run `filename.split('.')` the result is a list with two elements: ['corn\_riley\_2024', 'csv']

```
# Replace characteres
print(filename.replace('_', '-'))
```

corn-riley-2024.csv

```
# Join strings using the `+` operator
filename = "myfile"
extension = ".csv"
path = "/User/Documents/Datasets/"
```

```

fullpath_file = path + filename + extension
print(fullpath_file)

/User/Documents/Datasets/myfile.csv

# Find if word starts with one of the following sequences
print(base_filename.startswith(('corn','maize'))) # Note that the input is a tuple

# Find if word ends with one of the following sequences
print(base_filename.endswith(('2022','2023'))) # Note that the input is a tuple

True
False

# Passing variables into strings
station = 'Manhattan'
precip_amount = 25
precip_units = 'mm'

# Option 1 (preferred): f-string format (note the leading f)
option_1 = f"Today's Precipitation at the {station} station was {precip_amount} {precip_units}"
print(option_1)

# Option 2: %-string
# Note how much longer this syntax is. This also requires to keep track of the order of the variables
option_2 = "Today's Precipitation at the %s station was %s %s." % (station, precip_amount, precip_units)
print(option_2)

# ... however this syntax can sometimes be handy.
# Say you want to report parameter values using a label for one of your plots
par_values = [0.3, 0.1, 120] # Three parameter values, a list typically obtained by curve fitting
label = 'fit: a=%5.3f, b=%5.3f, c=%5.1f' % tuple(par_values)
print(label)

```

Today's Precipitation at the Manhattan station was 25 mm.  
 Today's Precipitation at the Manhattan station was 25 mm.  
 fit: a=0.300, b=0.100, c=120.0  
 DOY:A001  
 DOY:A365

```

# Formatting of values in strings
soil_pH = 6.7832
crop_name = "corn"

# Using an f-string to embed variables and format the pH value
message = f"The soil pH suitable for growing {crop_name} is {soil_pH:.2f}."

```

To specify the number of decimals for a value in an f-string, you can use the colon : followed by a format specifier inside the curly braces {}. For example, {variable:.2f} will format the variable to two decimal places. In this example, {soil\_pH:.2f} within the f-string takes the `soil_pH` variable and formats it to two decimal places. The :.2f part is the format specifier, where . indicates precision, 2 is the number of decimal places, and f denotes floating-point number formatting. This approach is highly efficient and readable, making f-strings a favorite among Python programmers.

```

# Say that you want to download data using the url from the NASA-MODIS satellite for a spe
doy = 1
print(f"DOY:A{doy:03d}")

```

DOY:A001

In the f-string `f"DOY:A{number:03d}"`, `{number:03d}` formats the variable `number`. The 03d specifier means that the number should be padded with zeros to make it three digits long (d stands for ‘decimal integer’ and 03 means ‘three digits wide, padded with zeros’). The letter ‘A’ is added as a prefix directly in the string. So, if `number` is 1, it gets formatted as 001, and the complete string becomes A001.

```

# Compare strings
print('loam' == 'Loam') # Returns False since case matters

print('loam' == 'Loam'.lower()) # Returns True since we convert the second word to lower c

```

False  
True

### Note

When comparing strings, using either `lower()` or `upper()` helps standardizing the strings before the boolean operation. This is particularly useful if you need to request information from users or deal with messy datasets that are not consistent.

## 14.4 Booleans

Boolean data types represent one of two values: `True` or `False`. Booleans are particularly powerful when used with conditional statements like `if`. By evaluating a boolean expression in an `if` statement, you can control the flow of your program, allowing it to make decisions and execute different code based on certain conditions. For instance, an `if` statement can check if a condition is `True`, and only then execute a specific block of code. Check the section about `if` statements for some examples.

**Boolean logical operators** `or`: Will evaluate to `True` if at least one (but not necessarily both) statements is `True` `and`: Will evaluate to `True` only if both statements are `True` `not`: Reverses the result of the statement

**Boolean comparison operators** `==`: equal `!=`: not equal `>=`: greater or equal than `<=`: less or equal than `>`: greater than `<`: less than

 Note

Python evaluates conditional arguments from left to right. The evaluation halts as soon as the outcome is determined, and the resulting value is returned. Python does not evaluate subsequent operands unless it is necessary to resolve the result.

```
# Example boolean logical operator

adequate_moisture = True
print(adequate_moisture)
print(type(adequate_moisture))

True
<class 'bool'>

# Example boolean comparison operators
optimal_moisture_level = 30 # optimal soil moisture level as a percentage
current_moisture_level = 25 # current soil moisture level as a percentage

is_moisture_optimal = current_moisture_level >= optimal_moisture_level
print(is_moisture_optimal)

True
```

```
chance_rain_tonight = 10 # probability of rainfall as a percentage  
  
water_plants = (current_moisture_level >= optimal_moisture_level) and (chance_rain_tonight  
print(water_plants)
```

```
True
```

## 14.5 Conversion between data types

In Python, converting between different data types, a process known as type casting, is a common and straightforward operation. You can convert data types using built-in functions like `int()`, `float()`, `str()`, and `bool()`. For instance, `int()` can change a floating-point number or a string into an integer, `float()` can turn an integer or string into a floating-point number, and `str()` can convert an integer or float into a string. These conversions are especially useful when you need to perform operations that require specific data types, such as mathematical calculations or text manipulation. However, it's important to be mindful that attempting to convert incompatible types (like trying to turn a non-numeric string into a number) can lead to errors.

```
# Integers to string  
  
int_num = 8  
print(int_num)  
print(type(int_num)) # Print data type before conversion  
  
int_str = str(int_num)  
print(int_str)  
print(type(int_str)) # Print resulting data type  
  
8  
<class 'int'>  
8  
<class 'str'>  
  
# Floats to string  
  
float_num = 3.1415  
print(float_num)  
print(type(float_num)) # Print data type before conversion
```

```

float_str = str(float_num)
print(float_str)
print(type(float_str)) # Print resulting data type

3.1415
<class 'float'>
3.1415
<class 'str'>

# Strings to integers/floats

float_str = '3'
float_num = float(float_str)
print(float_num)
print(type(float_num))

# Check if string is numeric
float_str.isnumeric()

3.0
<class 'float'>

True

# Floats to integers
float_num = 4.9
int_num = int(float_num)

print(int_num)
print(type(int_num))

4
<class 'int'>

```

In some cases Python will change the class according to the operation. For instance, the following code starts from two integers and results in a floating point.

```
numerator = 5
denominator = 2
print(type(numerator))
print(type(denominator))

answer = numerator / denominator # Two integers
print(answer)
print(type(answer)) # Result is a float
```

```
<class 'int'>
<class 'int'>
2.5
<class 'float'>
```

# 15 Data structures

In the realm of programming and data science, data structures act as containers where information is stored for later use. Python offers a variety of built-in data structures like lists, tuples, sets, and dictionaries, each with its own unique properties and use cases.

The choice of the right data structure often depends on factors like scalability, data format, data complexity, and the programmer's preference. Consider devoting some time before starting a new script to test and select an appropriate data structure for your program.

## 15.1 Lists

Lists are versatile data structures defined by square brackets [ ] that ideal for storing sequences of elements, such as strings, numbers, or a mix of different data types. **Lists are mutable**, meaning that you can modify their content. Lists also support nesting, where a list can contain other lists. A key feature of lists is the ability to access elements through indexing (for single item) or slicing (for multiple items). While similar to arrays in other languages, like Matlab, it's important to note that Python lists do not natively support element-wise operations, a functionality that is characteristic of NumPy arrays, a more advanced module that we will explore later.

```
# List with same data type
soil_texture = ["Sand", "Loam", "Silty clay", "Silt loam", "Silt"] # Strings (soil texture)
mean_sand = [92, 40, 5, 20, 5] # Integers (percent sand for each soil textural class)

print(soil_texture)
print(type(soil_texture)) # Print type of data structure

# List with mixed data types (strings, floats, and an entire dictionary)
# Sample ID, soil texture, pH value, and multiple nutrient concentration in ppm
soil_sample = ["Sample_001", "Loam", 6.5, {"N": 20, "P": 15, "K": 5}]

['Sand', 'Loam', 'Silty clay', 'Silt loam', 'Silt']
<class 'list'>
```

```
# Indexing a list
print(soil_texture[0]) # Accesses the first item
print(soil_sample[2])
```

Sand

6.5

```
# Slicing a list
print(soil_texture[2:4])
print(soil_texture[2:])
print(soil_texture[:3])
```

```
['Silty clay', 'Silt loam']
['Silty clay', 'Silt loam', 'Silt']
['Sand', 'Loam', 'Silty clay']
```

```
# Find the length of a list
print(len(soil_texture)) # Returns the number of items
```

5

### Note

Can you guess how many items are in the `soil_sample` list? Use Python to check your answer!

```
# Append elements to a list
soil_texture.append("Clay") # Adds 'Barley' to the list 'crops'
print(soil_texture)
```

```
['Sand', 'Loam', 'Silty clay', 'Silt loam', 'Silt', 'Clay']
```

```
# Append multiple elements
soil_texture.extend(["Loamy sand", "Sandy loam"])
print(soil_texture)
```

```
['Sand', 'Loam', 'Silty clay', 'Silt loam', 'Silt', 'Clay', 'Loamy sand', 'Sandy loam']
```

### Note

Appending multiple items using the `append()` method will result in nested lists, while using the `extend()` method will result in merged lists. Give it a try and see if you can observe the difference.

```
# Remove list element
soil_texture.remove("Clay")
print(soil_texture)
```

```
['Sand', 'Loam', 'Silty clay', 'Silt loam', 'Silt', 'Loamy sand', 'Sandy loam']
```

```
# Insert an item at a specified position or index
soil_texture.insert(2, "Clay") # Inserts 'Clay' back again, but at index 2
print(soil_texture)
```

```
['Sand', 'Loam', 'Clay', 'Silty clay', 'Silt loam', 'Silt', 'Loamy sand', 'Sandy loam']
```

```
# Remove element based on index
soil_texture.pop(4)
print(soil_texture)
```

```
['Sand', 'Loam', 'Clay', 'Silty clay', 'Silt', 'Loamy sand', 'Sandy loam']
```

```
# An alternative method to delete one or more elements of the list.
del soil_texture[1:3]
print(soil_texture)
```

```
['Sand', 'Silty clay', 'Silt', 'Loamy sand', 'Sandy loam']
```

## 15.2 Tuples

Tuples are an efficient data structure defined by parentheses ( ), and are especially useful for storing fixed sets of elements like coordinates in a two-dimensional plane (e.g., `point(x, y)`) or triplets of color values in the RGB color space (e.g., `(r, g, b)`). While tuples can be nested within lists and support operations similar to lists, like indexing and slicing, the

main difference is that **tuples are immutable**. Once a tuple is created, its content cannot be changed. This makes tuples particularly valuable for storing critical information that must remain constant in your code.

```
# Geographic coordinates
mauna_loa = (19.536111, -155.576111, 3397) # Mauna Load Observatory in Hawaii, USA
konza_prairie = (39.106704, -96.608968, 320) # Konza Prairie in Kansas, USA

locations = [mauna_loa, konza_prairie]
print(locations)

[(19.536111, -155.576111, 3397), (39.106704, -96.608968, 320)]
```

```
# A list of tuples
colors = [(0,0,0), (255,255,255), (0,255,0)] # Each tuple refers to black, white, and green
print(colors)
print(type(colors[0]))
```

```
[(0, 0, 0), (255, 255, 255), (0, 255, 0)]
<class 'tuple'>
```

### Note

What happens if we want to change the first element of the third tuple from 0 to 255?  
Hint: `colors[2][0] = 255`

## 15.3 Dictionaries

Dictionaries are a highly versatile and popular data structure that have the peculiar ability to store and retrieve data using key-value pairs defined within curly braces {} or using the `dict()` function. This means that you can access, add, or modify data using unique keys, making dictionaries incredibly efficient for organizing and handling data using named references.

Dictionaries are particularly useful in situations where data doesn't fit neatly into a matrix or table format and has multiple attributes, such as weather data, where you might store various weather parameters (temperature, humidity, wind speed) using descriptive keys. Unlike lists or tuples, dictionaries aren't ordered by nature, but they excel in scenarios where each piece of data needs to be associated with a specific identifier. This structure provides a straightforward and intuitive way to manage complex, unstructured data.

```

# Weather data is often stored in dictionary or dictionary-like data structures.
D = {'city':'Manhattan',
      'state':'Kansas',
      'coords': (39.208722, -96.592248, 350),
      'data': [ {'date' : '20220101',
                 'precipitation' : {'value':12.5, 'unit':'mm', 'instrument':'TE525'},
                 'air_temperature' : {'value':5.6, 'units':'Celsius', 'instrument':'ATMOS14'}
               },
                {'date' : '20220102',
                 'precipitation' : {'value':0, 'unit':'mm', 'instrument':'TE525'},
                 'air_temperature' : {'value':1.3, 'units':'Celsius', 'instrument':'ATMOS14'}
               }
            ]
}

print(D)
print(type(D))

```

{'city': 'Manhattan', 'state': 'Kansas', 'coords': (39.208722, -96.592248, 350), 'data': [{}  
<class 'dict'>

The example above has several interesting features:

- The city and state names are ordinary strings
- The geographic coordinates (latitude, longitude, and elevation) are grouped using a tuple.
- Weather data for each day is a list of dictionaries
- In a single dictionary we have observations for a given timestamp together with the associated metadata including units, sensors, and location.

Personally I think that dictionaries are ideal data structures in the context of reproducible science.

#### Note

It's important that you realize that the organization of the dictionary above depends on programmer preferences. For instance, rather than grouping all three coordinates into a tuple, a different programmer may prefer to store the values under individual `name:value` pairs, such as: `latitude : 39.208722, longitude : -96.592248, altitude : 350`

## 15.4 Sets

Sets are a unique and somewhat less commonly used data structure compared to lists, tuples, and dictionaries. Sets are defined with curly braces {} (without defining key-value pairs) or the `set()` function and are similar to mathematical sets, meaning they store unordered collections of unique items. In other words, Sets don't allow for duplicate items, items cannot

be changed (although items can be added and removed), and items are not indexed. This makes sets ideal for operations like determining membership, eliminating duplicates, and performing mathematical set operations such as unions, intersections, and differences. In scenarios like database querying or data analysis where you need to compare different datasets, sets can be used to find common elements (intersection), all elements (union), or differences between datasets.

```
# Union operation
field1_weeds = set(["Dandelion", "Crabgrass", "Thistle", "Dandelion"])
field2_weeds = set(["Thistle", "Crabgrass", "Foxtail"])
unique_weeds = field1_weeds.union(field2_weeds)
print(unique_weeds)

{'Thistle', 'Crabgrass', 'Dandelion', 'Foxtail'}

# Intersection operation
common_weeds = field1_weeds.intersection(field2_weeds)
print(common_weeds)

{'Thistle', 'Crabgrass'}

# Difference operation
different_weeds_in_field1 = field1_weeds.difference(field2_weeds)
print(different_weeds_in_field1)

different_weeds_in_field2 = field2_weeds.difference(field1_weeds)
print(different_weeds_in_field2)

{'Dandelion'}
{'Foxtail'}

# We can also chain more variables if needed
field3_weeds = set(["Pigweed", "Clover"])
field1_weeds.union(field2_weeds).union(field3_weeds)

{'Clover', 'Crabgrass', 'Dandelion', 'Foxtail', 'Pigweed', 'Thistle'}
```

**i** Note

This exercise with `sets` could be a good example of how to quickly merge field notes about dominant weed species from multiple agronomists visiting fields in a given region.

## 15.5 Practice

1. Create a list with the scientific names of three common grasses in the US Great Plains: big bluestem, switchgrass, indian grass, and little bluestem.
2. Using a periodic table, store in a dictionary the name, symbol, atomic mass, melting point, and boiling point of oxygen, nitrogen, phosphorus, and hydrogen. Then, write two separate python statements to retrieve the boiling point of oxygen and hydrogen. Combined, these two atoms can form water, which has a boiling point of 100 degrees Celsius. How does this value compare to the boiling point of the individual elements?
3. Without editing the dictionary that you created in the previous point, append the properties for a new element: carbon.
4. Create a list of tuples encoding the latitude, longitude, and altitude of three national parks of your choice.

# 16 Working with dates and times

Common tasks in crop and soil sciences usually require dealing with temporal information in the form of timestamps. Whether is the date a soil sample was collected, the day a crop was planted, or the timestamps recorded by dataloggers monitoring the weather, working with dates and times is a ubiquitous task in agronomy and related fields.

Python's `datetime` module is an excellent and versatile tool for handling dates and times. Here, we'll explore its syntax and functionality with practical examples.

## 16.0.1 List of shorthands to represent date and time components

%d = two-digit day %m = two-digit month %Y = two-digit year %H = two-digit hour %M = two-digit minute %S = two-digit second %b = three-letter month %a = Three-letter day of the week  
%A = Full day of the week

```
# Import module
from datetime import datetime, timedelta
```

## 16.1 Basic datetime Syntax

```
# Current date and time
now = datetime.now()
print(type(now)) # show data type
print("Current Date and Time:", now)

# Specific date and time
planting_date = datetime(2022, 4, 15, 8, 30)
print("Planting Date and Time:", planting_date.strftime('%Y-%m-%d %H:%M:%S'))

# Parsing String to Datetime
harvest_date = datetime.strptime("2022-09-15 14:00:00", '%Y-%m-%d %H:%M:%S')
print("Harvest Date:", harvest_date)
```

```
# We can also add custom datetime format in f-strings
print(f"Harvest Date: {harvest_date:%A, %B %d, %Y}")
```

```
<class 'datetime.datetime'>
Current Date and Time: 2024-01-12 16:56:03.399852
Planting Date and Time: 2022-04-15 08:30:00
Harvest Date: 2022-09-15 14:00:00
Harvest Date: Thursday, September 15, 2022
```

## 16.2 Working with timedelta

```
# Difference between two dates
duration = harvest_date - planting_date
print("Days Between Planting and Harvest:", duration.days)

# Total seconds of the duration
print("Total Seconds:", duration.total_seconds())

# Use total seconds to compute number of hours
print("Total hours:", round(duration.total_seconds()/3600), 'hours') # 3600 seconds per hour

# Adding ten days
emergence_date = planting_date + timedelta(days=10)
print("Crop emergence was on:", emergence_date)
```

```
Days Between Planting and Harvest: 153
Total Seconds: 13239000.0
Total hours: 3678 hours
Crop emergence was on: 2022-04-25 08:30:00
```

## 16.3 Using Pandas for datetime operations

```
# Import module
import pandas as pd
```

```

# Create a DataFrame with dates
df = pd.DataFrame({
    "planting_dates": ["2020-04-15", "2021-04-25", "2022-04-7"],
    "harvest_dates": ["2020-09-15", "2021-10-1", "2022-09-25"]
})

# Convert string to datetime in Pandas
df['planting_dates'] = pd.to_datetime(df['planting_dates'], format='%Y-%m-%d')
df['harvest_dates'] = pd.to_datetime(df['harvest_dates'], format='%Y-%m-%d')

# Add a timedelta column
df['growing_season_length'] = df['harvest_dates'] - df['planting_dates']

# Display dataframe
df.head()

```

|   | planting_dates | harvest_dates | growing_season_length |
|---|----------------|---------------|-----------------------|
| 0 | 2020-04-15     | 2020-09-15    | 153 days              |
| 1 | 2021-04-25     | 2021-10-01    | 159 days              |
| 2 | 2022-04-07     | 2022-09-25    | 171 days              |

# 17 Indexing and slicing

Indexing and slicing are fundamental concepts used to retrieve and modify data within sequences like strings, lists, and arrays. Python uses zero-based indexing, meaning that the first element of any sequence is accessed with index 0, not 1. **Indexing** allows you to access individual elements, while **slicing** lets you access a subset, or a slice, of a sequence.

## 17.1 Syntax for indexing and slicing one-dimensional arrays

```
# Indexing a one-dimensional array
element = array[index]

# Slicing a one-dimensional array
sub_array = array[start_index:end_index:step]
```

Omitting `start_index` (e.g., `array[:end_index]`) slices from the beginning to `end_index`.

Omitting `end_index` (e.g., `array[start_index:]`) slices from `start_index` to the end of the array.

```
# Generate integers from 0 to the specified number (non-inclusive)
numbers = list(range(10))
print(numbers)

# Find the first element of the list (indexing operation)
print(numbers[0])

# First and second element
print(numbers[0:2])

# All elements (from 0 and on)
print(numbers[0:])

# Every other element (specifying the total number of element)
print(numbers[0:10:2])
```

```

# Every other element (without specifying the total number of elements)
print(numbers[0:-1:2])

# Print the first 3 elements
print(numbers[:3])

# Slice from the 4th to the next-to-last element
print(numbers[4:-1])

# Print the last item of the list
print(numbers[-1])
print(numbers[len(numbers)-1])

```

```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0
[0, 1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
[0, 2, 4, 6, 8]
[0, 1, 2]
[4, 5, 6, 7, 8]
9
9

```

## 17.2 Syntax for indexing and slicing two-dimensional arrays

```

# Indexing a two-dimensional array
element = array[row_index][column_index]

# Slicing a two-dimensional array (with step)
sub_array = array[row_start:row_end:row_step, column_start:column_end:column_step]

```

Omitting `row_start` (e.g., `array[:row_end, :]`) slices from the beginning to `row_end` in all columns. Here “all columns” is represented by the `:` operator. Similarly you can use `:` to represent all rows.

Omitting `row_end` (e.g., `array[row_start:, :]`) slices from `row_start` to the end in all columns.

```

import numpy as np

# Indexing and slicing a two-dimensional array
np.random.seed(0)
M = np.random.randint(0,10,[5,5])
print(M)

# Write five python commands to obtain:
# top row
# bottom row
# right-most column
# left-most column
# upper-right 3x3 matrix

```

```

[[5 0 3 3 7]
 [9 3 5 2 4]
 [7 6 8 8 1]
 [6 7 7 8 1]
 [5 9 8 9 4]]

```

```

# Solutions

# Top row
print('Top row')
print(M[0,:]) # Preferred
print(M[:1][0])
print(M[0])

# Bottom row
print('Bottom row')
print(M[-1,:]) # Preferred
print(M[-1])
print(M[4,:]) # Requires knowing size of array in advance

# Right-most column
print('Right-most column')
print(M[:, -1])

# Left-most column
print('Left-most column')
print(M[:, 0])

```

```

# Upper-right 3x3 matrix
print('Upper 3x3 matrix')
print(M[0:3,M.shape[1]-3:M.shape[1]]) # More versatile
print(M[0:3,2:M.shape[1]])

# This: M[0:3,2:1] will not work

Top row
[5 0 3 3 7]
[5 0 3 3 7]
[5 0 3 3 7]
Bottom row
[5 9 8 9 4]
[5 9 8 9 4]
[5 9 8 9 4]
Right-most column
[7 4 1 1 4]
Left-most column
[5 9 7 6 5]
Upper 3x3 matrix
[[3 3 7]
 [5 2 4]
 [8 8 1]]
[[3 3 7]
 [5 2 4]
 [8 8 1]]

```

### 17.3 References

Source: <https://stackoverflow.com/questions/509211/understanding-slice-notation>

# 18 If statements

`if` statements control the flow of a program by evaluating whether a certain condition is true or false, and then executing specific actions based on that evaluation. The core syntax of an `if` statement involves specifying a condition and an action to be executed if that condition is true.

## 18.1 Syntax

```
if condition:  
    # Code block executes if condition is true  
elif another_condition:  
    # Code block executes if another condition is true  
else:  
    # Code block executes if none of the above conditions are true
```

## 18.2 Example 1: Saline, sodic, and saline-sodic soils

For instance, in soil science we can use `if` statements to categorize soils into saline, saline-sodic, or sodic based on electrical conductivity (EC) and the Sodium Adsorption Ratio (SAR). Sometimes soil pH is also a component of this classification, but to keep it simple we will only use EC and SAR for this example. We can write a program that uses `if` statements to classify salt-affected soils following this widely-used table:

| Soil Type    | EC         | SAR  |
|--------------|------------|------|
| Normal       | < 4.0 dS/m | < 13 |
| Saline       | 4.0 dS/m   | < 13 |
| Sodic        | < 4.0 dS/m | 13   |
| Saline-Sodic | 4.0 dS/m   | 13   |

[Here](#) is a great fact sheet where you can learn more about saline, sodic, and saline-sodic soils

```

ec = 3 # electrical conductivity in dS/m
sar = 16 # sodium adsorption ratio (dimensionless)

# Determine the category of soil
if (ec >= 4.0) and (sar < 13):
    classification = "Saline"

elif (ec < 4.0) and (sar >= 13):
    classification = "Sodic"

elif (ec >= 4.0) and (sar >= 13):
    classification = "Saline-Sodic"

else:
    classification = "Normal"

print(f'Soil is {classification}')

```

Soil is Sodic

### 18.3 Example 2: Climate classification based on aridity index

To understand global climate variations, one useful metric is the Aridity Index (AI), which compares annual precipitation to atmospheric demand. Essentially, a lower AI indicates a drier region:

$$AI = \frac{P}{PET}$$

where  $P$  is the annual precipitation in mm and  $PET$  is the annual cumulative potential evapotranspiration in mm.

Over time, the definition of AI has evolved, leading to various classifications in the literature. Below is a simplified summary of these classifications:

| Climate class | Value              |
|---------------|--------------------|
| Hyper-arid    | $0.03 < AI$        |
| Arid          | $0.03 < AI < 0.20$ |
| Semi-arid     | $0.20 < AI < 0.50$ |

| Climate class | Value                 |
|---------------|-----------------------|
| Dry sub-humid | $0.50 < AI \leq 0.65$ |
| Sub-humid     | $0.65 < AI \leq 0.75$ |
| Humid         | $AI > 0.75$           |

```

# Define annual precipitation and atmospheric demand for a location
P = 1200    # mm per year
PET = 1800 # mm per year

# Compute Aridity Index
AI = P/PET

# Find climate class
if AI <= 0.03:
    climate_class = 'Arid'

elif AI > 0.03 and AI <= 0.2:
    climate_class = 'Arid'

elif AI > 0.2 and AI <= 0.5:
    climate_class = 'Semi-arid'

elif AI > 0.5 and AI <= 0.65:
    climate_class = 'Dry sub-humid'

elif AI > 0.65 and AI <= 0.75:
    climate_class = 'Sub-humid'

else:
    climate_class = 'Humid'

print('Climate classification for this location is:',climate_class,'(AI='+str(round(AI,2)))

```

Climate classification for this location is: Sub-humid (AI=0.67)

## 18.4 Comparative anatomy of If statements

### 18.4.0.0.1 Python

```
pH = 7 # Soil pH

if pH < 7:
    soil_class = 'Acidic'
elif pH == 7:
    soil_class = 'Neutral'
elif pH <= 14:
    soil_class = 'Alkaline'
else:
    print('pH value out of range.')

print(soil_class)
```

Output: Neutral

#### 18.4.0.0.2 Matlab

```
pH = 7; % Soil pH

if pH < 7
    soil_class = 'Acidic';
elseif pH == 7
    soil_class = 'Neutral';
elseif pH <= 14
    soil_class = 'Alkaline';
else
    disp('pH value out of range.');
end

disp(soil_class);
```

Output: Neutral

#### 18.4.0.0.3 Julia

```
pH = 7 # Soil pH

if pH < 7
    soil_class = "Acidic"
elseif pH == 7
```

```

        soil_class = "Neutral"
elseif pH <= 14
    soil_class = "Alkaline"
else
    println("pH value out of range.")
end

println(soil_class)

```

**Output:** Neutral

#### 18.4.0.0.4 R

```

pH <- 7 # Soil pH

if (pH < 7) {
    soil_class <- 'Acidic'
} else if (pH == 7) {
    soil_class <- 'Neutral'
} else if (pH <= 14) {
    soil_class <- 'Alkaline'
} else {
    print('pH value out of range.')
}

print(soil_class)

```

**Output:** [1] "Neutral"

#### 18.4.0.0.5 JavaScript

```

let pH = 7; // Soil pH

let soil_class;
if (pH < 7) {
    soil_class = 'Acidic';
} else if (pH == 7) {
    soil_class = 'Neutral';
} else if (pH <= 14) {
    soil_class = 'Alkaline';
}

```

```

} else {
    console.log('pH value out of range.');
}

console.log(soil_class);

```

**Output:** Neutral

#### 18.4.0.0.6 Commonalities among programming languages:

- All languages use a conditional `if` keyword to start the statement.
- They employ logical conditions (e.g., `<`, `==`, `<=`) to evaluate true or false.
- The use of `else if` or its equivalent for additional conditions.
- The presence of `else` to handle cases that don't meet any `if` or `else if` conditions.
- Block of code under each condition is indented or enclosed (e.g., `{}` in JavaScript, `end` in Matlab and Julia).

## 18.5 References

Havlin, J. L., Tisdale, S. L., Nelson, W. L., & Beaton, J. D. (2016). Soil fertility and fertilizers. Pearson Education India.

Spinoni, J., Vogt, J., Naumann, G., Carrao, H. and Barbosa, P., 2015. Towards identifying areas at climatological risk of desertification using the Köppen–Geiger classification and FAO aridity index. International Journal of Climatology, 35(9), pp.2210-2222.

Zhang, H. 2017. Interpreting Soil Salinity Analyses. L-297. Oklahoma Cooperative Extension Service. [Link](#)

Zomer, R. J., Xu, J., & Trabucco, A. (2022). Version 3 of the global aridity index and potential evapotranspiration database. Scientific Data, 9(1), 409.

# 19 Functions

In this tutorial, we delve into the basics of functions, covering key concepts like function declaration, inputs, outputs, and documentation through docstrings. Functions are powerful tools, akin to a wrench or screwdriver in a toolbox, that are used for executing specific tasks within your code. Functions are essentially named encapsulated snippets of code that can be reused multiple times. Function help organizing your code, avoiding code repetition, and reduce errors. This reusability not only enhances the modularity of your current project, but also extends to future projects, allowing you to build a personal library of useful functions.

Utilizing functions involves two key steps:

- 1) define or declare the function, setting up what it does and how,
- 2) call or invoke the function whenever you need its functionality in your code.

This two-step process (defining and calling functions) is fundamental to programming and is crucial for creating well-structured, maintainable, and scalable code.

## 19.1 Syntax

This is the main syntax to define your own functions:

```
def function_name(parameters, par_opt=par_value):  
    # Code block  
    return result
```

Let's look at a few function examples to see this two-step process in action.

```
# Import necessary modules  
import numpy as np
```

## 19.2 Example function: Compute vapor pressure deficit

The vapor pressure deficit (VPD) represents the “thirst” of the atmosphere and is computed as the difference between the saturation vapor pressure and the actual vapor pressure. The

saturation vapor pressure can be accurately approximated as a function of air temperature using the empirical [Tetens equation](#). Here is the set equations to compute VPD:

**Saturation vapor pressure:**

$$e_{sat} = 0.611 \exp\left(\frac{17.502 T}{T + 240.97}\right)$$

**Actual vapor pressure:**

$$e_{act} = e_{sat} \frac{RH}{100}$$

**Vapor pressure deficit:**

$$VPD = e_{sat} - e_{act}$$

**Variables**

$e_{sat}$  is the saturation vapor pressure deficit (kPa)

$e_{act}$  is the actual vapor pressure (kPa)

$VPD$  is the vapor pressure deficit (kPa)

$T$  is air temperature ( $^{\circ}\text{C}$ )

$RH$  is relative humidity (%)

### 19.2.1 Define function

In the following example we will focus on the main building blocks of a function, but we will ignore error handling and checks to ensure that inputs have the proper data type. For more details on how to properly handle errors and ensure inputs have the correct data type see the error handling tutorial.

```
# Define function

def compute_vpd(T, RH, unit='kPa'):
    """
    Function that computes the air vapor pressure deficit (vpd).

    Parameters:
    T (integer, float): Air temperature in degrees Celsius.
    RH (integer, float): Air relative humidity in percentage.
    unit (string): Unit of the output vpd value.
        One of the following: kPa (default), bars, psi
```

```

>Returns:
float: Vapor pressure deficit in kiloPascals (kPa).

>Authors:
Andres Patrignani

>Date created:
6 January 2024

>Reference:
Campbell, G. S., & Norman, J. M. (2000).
An introduction to environmental biophysics. Springer Science & Business Media.
"""

# Compute saturation vapor pressure
e_sat = 0.611 * np.exp((17.502*T) / (T + 240.97)) # kPa

# Compute actual vapor pressure
e_act = e_sat * RH/100 # kPa

# Compute vapor pressure deficit
vpd = e_sat - e_act # kPa

# Change units if necessary
if unit == 'bars':
    vpd *= 0.01 # Same as vpd = vpd * 0.01

elif unit == 'psi':
    vpd *= 0.1450377377 # Convert to pounds per square inch (psi)

return vpd

```

### Syntax note

Did you notice the expression `vpd *= 0.01`? This is a compact way in Python to do `vpd = vpd * 0.01`. You can also use it with other operators, like `+=` for adding or `-=` for subtracting values from a variable.

## 19.2.2 Description of function components

1. **Function Definition:** `def compute_vpd(T, RH, unit='kPa'):` This line defines the function with the name `compute_vpd`, which takes two parameters, `T` for temperature and `RH` for relative humidity. The function also includes an optional argument `unit=` that has a default value of `kPa`.

### 2. Docstring:

```
"""
Function that computes the air vapor pressure deficit (vpd).

Parameters:
T (integer, float): Air temperature in degrees Celsius.
RH (integer, float): Air relative humidity in percentage.
unit (string): Unit of the output vpd value.
    One of the following: kPa (default), bars, psi

Returns:
float: Vapor pressure deficit in kiloPascals (kPa).

Authors:
Andres Patrignani

Date created:
6 January 2024

Reference:
Campbell, G. S., & Norman, J. M. (2000).
An introduction to environmental biophysics. Springer Science & Business Media.
"""
```

The triple-quoted string right after the function definition is the docstring. It provides a brief description of the function, its parameters, their data types, and what the function returns.

### 3. Saturation Vapor Pressure Calculation:

```
e_sat = 0.611 * np.exp((17.502*T) / (T + 240.97)) # kPa
```

This line of code calculates the saturation vapor pressure (`e_sat`) using air temperature `T`. It's a mathematical expression that uses the `exp` function from the NumPy library (`np`), which should be imported at the beginning of the script.

#### 4. Actual Vapor Pressure Calculation:

```
e_act = e_sat * RH/100 # kPa
```

This line calculates the actual vapor pressure (`e_act`) based on the saturation vapor pressure and the relative humidity `RH` of air.

#### 5. Vapor Pressure Deficit Calculation:

```
vpd = e_sat - e_act # kPa
```

Here, the vapor pressure deficit (`vpd`) is computed by subtracting the actual vapor pressure from the saturation vapor pressure.

#### 6. Unit conversion:

```
# Change units if necessary
if unit == 'bars':
    vpd = vpd * 0.01

elif unit == 'psi':
    vpd = vpd * 0.1450377377 # Convert to pounds per square inch (psi)
```

In this step we change the units of the resulting `vpd` before returning the output. Note that since the value of `vpd` using the equations in the function is already in `kPa`, so there is no need to handle this scenario in the `if` statement.

#### 7. Return Statement:

```
return vpd
```

The `return` statement sends back the result of the function (`vpd`) to wherever the function was called.

##### Note

In Python functions, you can use optional parameters with default values for flexibility, placing them after mandatory parameters in the function's definition.

### 19.2.3 Call function

Having named our function and defined its inputs, we can now invoke the function without duplicating the code.

```
# Define input variables
T = 25 # degrees Celsius
RH = 75 # percentage

# Call the function (without using the optional argument)
vpd = compute_vpd(T, RH)

# Display variable value
print(f'The vapor pressure deficit is {vpd:.2f} kPa')
```

The vapor pressure deficit is 0.79 kPa

```
# Call the function using the optional argument to specify the unit in `bars`
vpd = compute_vpd(T, RH, unit='bars')

# Display variable value
print(f'The vapor pressure deficit is {vpd:.3f} bars')
```

The vapor pressure deficit is 0.008 bars

```
# Call the function using the optional argument to specify the unit in `psi`
vpd = compute_vpd(T, RH, unit='psi')

# Display variable value
print(f'The vapor pressure deficit is {vpd:.3f} psi')
```

The vapor pressure deficit is 0.115 psi

### ! Important

In Python, the sequence in which you pass input arguments into a function is critical because the function expects them in the order they were defined. If you call `compute_vpd` with the inputs in the wrong order, like `compute_vpd(RH, T)`, the function will still execute, but it will use relative humidity (`RH`) as temperature and temperature (`T`) as humidity, leading to incorrect results. To ensure accuracy, you must match the order to the function's definition: `compute_vpd(T, RH)`.

#### 19.2.4 Evaluate function performance

Code performance in terms of execution time directly impacts the data analysis and visualization experience. The `perf_counter()` method within the `time` module provides a high-resolution timer that can be used to track the execution time of your code, offering a precise measure of performance. By recording the time immediately before and after a block of code runs, and then calculating the difference, `perf_counter()` helps you understand how long your code takes to execute. This is particularly useful for optimizing code and identifying bottlenecks in your Python programs. However, it is important to balance performance with the principle that premature optimization in the early stages of a project is often counterproductive. Optimization should come at a later stage when the code is correct and its performance bottlenecks are clearly identified.

```
# Import time module
import time

# Get initial time
tic = time.perf_counter()

vpd = compute_vpd(T, RH, unit='bars')

# Get final time
toc = time.perf_counter()

# Compute elapsed time
elapsed_time = toc - tic
print("Elapsed time:", elapsed_time, "seconds")
```

Elapsed time: 7.727195043116808e-05 seconds

#### 19.2.5 Access function help (the docstring)

```
compute_vpd?
```

```
Signature: compute_vpd(T, RH, unit='kPa')
Docstring:
Function that computes the air vapor pressure deficit (vpd).

Parameters:
T (integer, float): Air temperature in degrees Celsius.
```

```
RH (integer, float): Air relative humidity in percentage.  
unit (string): Unit of the output vpd value.  
    One of the following: kPa (default), bars, psi
```

Returns:

```
float: Vapor pressure deficit in kiloPascals (kPa).
```

Authors:

```
Andres Patrignani
```

Date created:

```
6 January 2024
```

Reference:

```
Campbell, G. S., & Norman, J. M. (2000).
```

```
An introduction to environmental biophysics. Springer Science & Business Media.
```

```
File:      /var/folders/w1/cgh8d8y962g9c6p4_dxgbn2jh5jy11/T/ipykernel_40431/2839097177.py
```

```
Type:      function
```

## 19.3 Function variable scope

One aspect of Python functions that we did not cover is variable scope. In Python, variables defined inside a function are local to that function and can't be accessed from outside of it, while variables defined outside of functions are global and can be accessed from anywhere in the script. It's like having a conversation in a private room (function) versus a public area (global scope).

To prevent confusion, it's best to follow good naming conventions for variables in your scripts. However, in extensive scripts with numerous functions—both written by you and imported from other modules—tracking every variable name can be challenging. This is where the local variable scope of Python functions comes to rescue, ensuring that variables within a function don't interfere with those outside.

Below are a few examples to practice and consider.

```
# Example: Access a global variable from inside of a function  
variable_outside = 1 # Accessible from anywhere in the script  
  
def my_function():  
    variable_inside = 2 # Only accessible within this function (it's not being used for anything)  
    print(variable_outside)
```

```

# Invoke the function
my_function()

1

# Example: Modify global variable from inside of a function (will not work)
# See next example for a working solution
variable_outside = 1 # Accessible from anywhere in the script

def my_function():
    variable_outside += 5 # If you mute this line with a comment `#`, it works fine.
    print(variable_outside)

# Invoke the function
my_function()

# See, Python is allowing us to print `variable_outside`, but as soon as we
# request to perform an operation on it, it searches in the local workspace of the function
# for a variable called `variable_outside`, but since this variable has not been defined
# WITHIN the function, then it throws an error.
# This is a good thing, and a safety net to prevent costly mistakes.

```

```

UnboundLocalError: local variable 'variable_outside' referenced before assignment

# Example: Modify global variables inside of a function
# This is an advance feature that I would advice against when learning how to code

variable_outside = 1 # Accessible from anywhere in the script

def my_function():
    global variable_outside # We tell Python to use the variable defined outside in the ne

    variable_outside += 5 # If you mute this line with a comment `#`, it works fine.
    print(variable_outside)

# Invoke the function
my_function() # The function changes the value of the variable_outside

# Print the value of the variable

```

```
print(variable_outside) # Same value as before since we changed it inside the function
```

```
6  
6
```

```
# Example: A global and a local variable with the same name
```

```
variable_outside = 1 # Accessible from anywhere in the script
```

```
def my_function():
```

```
    variable_outside = 1 # A different variable with the same name. Only available inside
    variable_outside += 5 # We are changing the value in the previous line, not the one defined outside
    print(variable_outside)
```

```
# Invoke the function
```

```
my_function() # This prints the variable inside the function
```

```
# This prints the variable we defined at the top, which remains unchanged
print(variable_outside)
```

```
6  
1
```

## 19.4 Python handy functions: map, filter, and reduce

### 19.4.1 map

**Description:** Applies a given function to each item of an iterable (like a list) and returns a map object. **Use-case:** Transforming data elements in a collection.

```
celsius = [0,10,20,100]
fahrenheit = list(map(lambda x: (x*9/5)+32, celsius))
print(fahrenheit)
```

```
[32.0, 50.0, 68.0, 212.0]
```

```

# Convert a DNA sequence into RNA
# Remember that RNA contains uracil instead of thymine
dna = 'ATTCGGGCAAATATGC'
lookup = dict({"A":"U", "T":"A", "C":"G", "G":"C"})
rna = list(map(lambda x: lookup[x], dna))
print(''.join(rna))

```

UAAGCCCCUUUAUACG

### 19.4.2 filter

**Description:** Filters elements of an iterable based on a function that tests each element.  
**Use-case:** Selecting elements that meet specific criteria.

```

# Get the occurrence of all adenine nucleotides
dna = 'ATTCGGGCAAATATGC'
list(filter(lambda x: x == "A", dna))

['A', 'A', 'A', 'A', 'A']

# Find compacted soils
bulk_densities = [1.01, 1.52, 1.84, 1.45, 1.32]
compacted_soils = list(filter(lambda x: x > 1.6, bulk_densities))
print(compacted_soils)

```

[1.84]

```

# Find hydrophobic soils based on regular function
def is_hydrophobic(contact_angle):
    """
    Function that determines whether a soil is hydrophobic
    based on its contact angle.
    """
    if contact_angle < 90:
        repel = False
    elif contact_angle >= 90 and contact_angle <= 180:
        repel = True

```

```
    return repel

contact_angles = [5,10,20,50,90,150]
list(filter(is_hydrophobic, contact_angles))
```

```
[90, 150]
```

### 19.4.3 reduce

**Description:** Applies a function cumulatively to the items of an iterable, reducing the iterable to a single value. **Use-case:** Aggregating data elements.

```
from functools import reduce

# Compute total yield
crop_yields = [1200, 1500, 1800, 2000]
total_yield = reduce(lambda x, y: x + y, crop_yields)
print(total_yield)
```

```
6500
```

#### i Note

While the `map`, `filter`, and `reduce` functions are useful in standard Python, the functions are less critical when working with Pandas or NumPy, as these libraries already provide built-in, optimized methods for element-wise operations and data manipulation. Numpy typically surpasses the need for `map`, `filter`, or `reduce` in most scenarios.

## 19.5 Comparative anatomy of functions

### 19.5.0.0.1 Python

```
def hypotenuse(C1, C2):
    H = (C1**2 + C2**2)**0.5
    return H

hypotenuse(3, 4)
```

#### 19.5.0.0.2 Matlab

```
function H = hypotenuse(C1, C2)
    H = sqrt(C1^2 + C2^2);
end

hypotenuse(3, 4)
```

#### 19.5.0.0.3 Julia

```
function hypotenuse(C1, C2)
    H = sqrt(C1^2 + C2^2)
    return H
end

hypotenuse(3, 4)
```

#### 19.5.0.0.4 R

```
hypotenuse <- function(C1, C2) {
    H = sqrt(C1^2 + C2^2)
    return(H)
}

hypotenuse(3, 4)
```

#### 19.5.0.0.5 JavaScript

```
function hypotenuse(C1, C2) {
    let H = Math.sqrt(C1**2 + C2**2);
    return H;
}

hypotenuse(3, 4);
```

#### 19.5.0.0.6 Commonalities among programming languages:

- All languages use a keyword (like `function` or `def`) to define a function.

- They specify function names and accept parameters within parentheses.
- The body of the function is enclosed in a block (using braces {} like in JavaScript and R, indentation in the case of Python, or end in the case of Julia and Matlab).
- Return statements are used to output the result of the function.
- Functions are invoked by calling their name followed by arguments in parentheses.

## 19.6 Practice

1. Create a function that computes the amount of lime required to increase an acidic soil pH. You can find examples in most soil fertility textbooks or extension fact sheets from multiple land-grant universities in the U.S.
2. Create a function that determines the amount of nitrogen required by a crop based on the amount of nitrates available at pre-planting, a yield goal for your region, and the amount of nitrogen required to produce the the yield goal.
3. Create a function to compute the amount of water storage in the soil profile from inputs of volumetric water content and soil depth.
4. Create a function that accepts latitude and longitude coordinates in decimal degrees and returns the latitude and longitude values in [sexagesimal degrees](#). The function should accept Lat and Lon values as separate inputs e.g. `fun(lat,lon)` and must return a list of tuples with four components for each coordinate: degrees, minutes, seconds, and quadrant. The quadrant would be North/South for latitude and East/West for longitude. For instance: `fun(19.536111, -155.576111)` should result in `[(19,32,10,'N'),(155,34,34,'W')]`

# 20 Lambda functions

Lambda functions in Python offer an alternative way to create concise, one-line expressions for specific tasks, eliminating the need for a separate named function using the `def` keyword. They are characterized by their simplicity and ability to return a value directly without a `return` statement, making them perfect for quick, in-line operations like applying formulas in optimization routines or sorting and filtering data. Continuing with the analogy between regular functions and tools in a toolbox, think of lambda functions as the equivalent of using a kitchen knife or a coin in your pocket for quickly tightening a screw.

## 20.1 Syntax

```
lambda arguments: expression
```

Let's learn about Lambda functions by coding a few examples. We will start by importing the Numpy module, which we will need for some operations.

```
# Import modules
import numpy as np
```

## 20.2 Example 1: Convert degrees Fahrenheit to Celsius

Consider a scenario where you need to convert temperatures from degrees Fahrenheit to degrees Celsius frequently. Instead of repeatedly typing the conversion formula, you can encapsulate the expression in a lambda function for easy reuse. This approach not only avoids code repetition, but also reduces the risk of errors, as you write the formula just once and then call the lambda function by its name whenever needed.

$$C = \frac{5}{9}(F - 32)$$

where  $F$  is temperature in degrees Fahrenheit and  $C$  is the resulting temperature in degrees Celsius.

```

# Define the lambda function (note the bold green reserved word)
fahrenheit_to_Celsius = lambda F: 5/9 * (F-32)

# Call the function
F = 212 # temperature in degrees Fahrenheit
C = fahrenheit_to_Celsius(212)

print(f"A temperature of {F:.1f} °F is equivalent to {C:.1f} °C")

```

A temperature of 212.0 °F is equivalent to 100.0 °C

## 20.3 Breakdown of Lambda function components

Similar to what we did with regular functions, let's breakdown the components of a lambda function.

### 1. Defining the Lambda function:

```
fahrenheit_to_Celsius = lambda F: 5/9 * (F-32)
```

This line defines a lambda function named `fahrenheit_to_Celsius`.

### 2. Lambda keyword:

- `lambda`: This is the keyword that signifies the start of a lambda function in Python. It's followed by the parameters and the expression that makes up the function.

### 3. Parameter:

- `F`: This represents the parameter of the lambda function. In this case, `F` is the input temperature in degrees Fahrenheit that you want to convert to Celsius.

### 4. Function expression:

- `5/9*(F-32)`: This is the expression that gets executed when the lambda function is called. It's the formula for converting degrees Fahrenheit to Celsius.

#### Note

Note that, other than a simple line comment, `Lambda` functions offer very limited possibilities to provide associated documentation for the code, inputs, and outputs. If you need to a multi-line function or even a single-line function with detailed documentation,

then a regular Python function is the way to go.

## 20.4 Example 2: Estimate atmospheric pressure from altitude

Estimating atmospheric pressure from altitude (learn more about the topic [here](#)) is a classic problem in environmental science and meteorology, and using Python functions is an excellent tool for solving it.

$$P = 101.3 \exp\left(\frac{-h}{8400}\right)$$

where  $P$  is atmospheric pressure in  $kPa$  and  $h$  is altitude above sea level in meters. The coefficient 8400 is the result of aggregating the values for the gravitational acceleration, the molar mass of dry air, the universal gas constant, and sea level standard temperature; and converting from  $Pa$  to  $kPa$ .

```
# Define lambda function
estimate_atm_pressure = lambda A: 101.3 * np.exp(-A/8400) # atm pressure in kPa

# Compute atmospheric pressure for Kansas City, KS
city = "Kansas City, KS"
h = 280 # meters a.s.l.
P = estimate_atm_pressure(h)

print(f"Pressure in {city} at an elevation of {h} m is {P:.1f} kPa")
```

Pressure in Kansas City, KS at an elevation of 280 m is 98.0 kPa

```
# Compute atmospheric pressure for Denver, CO
city = "Denver, CO"
h = 1600 # meters a.s.l.
P = estimate_atm_pressure(h)

print(f"Pressure in {city} at an elevation of {h} m is {P:.1f} kPa.")
```

Pressure in Denver, CO at an elevation of 1600 m is 83.7 kPa.

## 20.5 Example 3: Volume of tree trunk

In the fields of forestry and ecology, trunk volume is a key metric for assessing and monitoring tree size. While tree trunks can have complex shapes, their general resemblance to cones allows for reasonably accurate volume estimations using basic measurements like trunk diameter and height. You can learn more about [measuring tree volumes here](#). To calculate trunk volume, we can use a simplified formula:

$$V = \frac{\pi h (D_1^2 + D_2^2 + D_1 D_2)}{12}$$

where  $D_1, D_2$  are the diameters of the top and bottom circular cross-sections of the tree.

Let's implement this formula using a Lambda function and then I propose computing the approximate volume of a Giant Sequoia. The *General Sherman*, located in the Sequoia National Park in California, is the largest single-stem living tree on Earth with an approximate height of 84 m and a diameter at breast height of about 7.7 m. This example for computing tree volumes illustrates the practical application of programming in environmental science.

```
compute_trunk_volume = lambda d1, d2, h: (np.pi * h * (d1**2 + d2**2 + d1*d2))/12

# Approximate tree dimensions (these values result
diameter_base = 7.7 # meters
diamter_top = 1 # meters (I assumed this value for the top of the stem)
height = 84 # meters

# Compute volume. The results is remarkably similar tot he reported 1,487 m^3
trunk_volume = compute_trunk_volume(diameter_base, diamter_top, height) # m^3

print(f"The trunk volume of a Giant Sequoia is {trunk_volume:.0f} cubic meters")

# Find relative volume compared to an Olympic-size (50m x 25m x 2m) swimming pool.
pool_volume = 50 * 25 * 2
rel_volume = trunk_volume/pool_volume*100 # Relative volume trunk/pool

# Ratio of volumes
print(f"Trunk volume is {rel_volume:.1f}% the size of an Olympic-size pool")
```

The trunk volume of a Giant Sequoia is 1495 cubic meters  
Trunk volume is 59.8% the size of an Olympic-size pool

## 20.6 Example 4: Calculate the sodium adsorption ratio (SAR).

The Sodium Adsorption Ratio (SAR) is a water quality indicator for determining the suitability for agricultural irrigation. In high concentrations, sodium has a negative impact on plant growth and disperses soil colloids, which has a detrimental impact on soil aggregation, soil structure and infiltration. It uses the concentrations of sodium, calcium, and magnesium ions measured in milliequivalents per liter (meq/L) to calculate the SAR value based on the formula:

$$SAR = \frac{\text{Na}^+}{\sqrt{\frac{\text{Ca}^{2+} + \text{Mg}^{2+}}{2}}}$$

```
# Define lambda function
calculate_sar = lambda na,ca,mg: na / ((ca+mg) / 2)**0.5

# Determine SAR value for a water sample with the following ion concentrations

na = 10 # Sodium ion concentration in meq/L
ca = 5 # Calcium ion concentration in meq/L
mg = 2 # Magnesium ion concentration in meq/L

sar_value = calculate_sar(na, ca, mg)
print(f"Sodium Adsorption Ratio (SAR) is: {sar_value:.1f}")

# SAR values <5 are typically excellent for irrigation,
# while SAR values >15 are typically unsuitable for irrigation of most crops.
```

Sodium Adsorption Ratio (SAR) is: 5.3

## 20.7 Example 5: Compute soil porosity

Soil porosity refers to the percentage of the soil's volume that is occupied by voids that can be occupied by air and water. Soil porosity is a soil physical property that conditions the soil's ability to hold and transmit water, heat, and air, and is essential for root growth and microbial activity. For a given textural class, soils with lower porosity values tend to be compacted compared to soils with greater porosity values.

Porosity ( $f$ , dimensionless) is defined as the volume of voids over the total volume of soil, and it can be approximated using bulk density ( $\rho_b$ ) and particle density ( $\rho_s$ , often assumed to have a value of  $2.65 \text{ g cm}^{-3}$  for mineral soils):

$$f = \left(1 - \frac{\rho_b}{\rho_s}\right)$$

```
# Define lambda function
# bd is the bulk density
calculate_porosity = lambda bd: 1 - (bd/2.65)

# Compute soil porosity
bd = 1.35 # Bulk density in g/cm^3
f = calculate_porosity(bd)

# Print result
print(f'The porosity of a soil with a bulk density of {bd:.2f} g/cm^3 is {f*100:.1f} %')
```

The porosity of a soil with a bulk density of 1.35 g/cm<sup>3</sup> is 49.1 %

## 20.8 Practice

- Implement Stoke's Law as a lambda function. Then, call the function to estimate the terminal velocity of a 1 mm diameter sand particle in water and a 1 mm raindrop in air.
- Implement a quadratic polynomial describing a yield-nitrogen response curve. Here is one model that you can try, but there are many empirical yield-N relationships in the literature:  $y = -0.0034x^2 + 0.9613x + 115.6$

# 21 Inputs

The `input()` function is a straightforward way to receive user input, making Python scripts interactive and adaptable to user-driven data. When `input()` is called, the program pauses and waits for the user to type something into the console. Once the user presses the `Enter` key, the input is read as a string (even if you entered a number). This function is particularly useful for interactive programs where user data is required.

To illustrate the `input()` function we will use code from other tutorials in this book to classify soils into saline, sodic, or saline-sodic soils, so that you can see how lessons in data types, functions, and control flow play together to create programs.

```
# ----- Request input from user -----
soil_location = input("Enter location where soil was collected: ") # Manhattan, KS
soil_ec = float(input("Enter the soil EC (dS/m): ")) # EC = 5.6 dS/m
soil_na = float(input("Enter the soil Na (meq/L: ")) # Na = 100 meq/L
soil_ca = float(input("Enter the soil Ca (meq/L: ")) # Ca = 20 meq/L
soil_mg = float(input("Enter the soil Mg (meq/L: ")) # Mg = 10 meq/L

# ----- Define functions -----
# Define lambda function to compute SAR
calculate_sar = lambda na,ca,mg: na / ((ca+mg) / 2)**0.5

# Define regular function for classifying soil based on EC and SAR
def classify_soil(ec,sar):
    if (ec >= 4.0) and (sar < 13):
        classification = "Saline"
    elif (ec < 4.0) and (sar >= 13):
        classification = "Sodic"
    elif (ec >= 4.0) and (sar >= 13):
        classification = "Saline-Sodic"
    else:
        classification = "Normal"
    return classification
```

```

# ----- Call functions -----
# Compute SAR calling lambd function
soil_sar = calculate_sar(soil_na, soil_ca, soil_mg)

# Classify soil calling regular function
soil_class = classify_soil(soil_ec, soil_sar)

# ----- Display results -----
# Print soil classification including EC and SAR values
print(f'Soil from {soil_location} is {soil_class} (EC={soil_ec:.1f} dS/m and SAR={soil_sar:.1f}')

```

Enter location where soil was collected: Manhattan, KS

Enter the soil EC (dS/m): 5.6

Enter the soil Na (meq/L: 100

Enter the soil Ca (meq/L: 20

Enter the soil Mg (meq/L: 10

Soil from Manhattan, KS is Saline-Sodic (EC=5.6 dS/m and SAR=25.8)

## 21.1 Practice

- Create a script that computes the soil bulk density, porosity, gravimetric water content, and volumetric water content given the mass of wet soil, mass of oven-dry soil, and the volume of the soil. Your code should contain a function that includes an optional input parameter for particle density with a default value of 2.65 g/cm<sup>3</sup>.

# 22 For loop

For loops are essential in programming for executing a block of code multiple times, automating repetitive tasks efficiently. They are particularly useful in data science for iterating through various data structures like lists and dictionaries. Unlike conventional counting that starts from 1, Python's for loops begin at index 0, iterating over sequences starting from the first element.

## 22.1 Syntax

```
for item in iterable:  
    # Code block to execute for each item
```

## 22.2 Example 1: Basic For loop

Suppose we have a list of soil nitrogen levels from different test sites and we want to print each value. Here's how you can do it:

```
# Example of a for loop  
  
# List of soil nitrogen levels in mg/kg  
nitrogen_levels = [15, 20, 10, 25, 18]  
  
# Iterating through the list  
for level in nitrogen_levels:  
    print(f"Soil Nitrogen Level: {level} mg/kg")
```

```
Soil Nitrogen Level: 15 mg/kg  
Soil Nitrogen Level: 20 mg/kg  
Soil Nitrogen Level: 10 mg/kg  
Soil Nitrogen Level: 25 mg/kg  
Soil Nitrogen Level: 18 mg/kg
```

## 22.3 Example 2: For loop using the enumerate function

The `enumerate` function adds a counter to the loop, providing the index position along with the value. This is helpful when you need to access the position of the elements as you iterate.

Let's modify the previous example to include the sample number using `enumerate`:

```
# Iterating through the list with enumerate
for index, level in enumerate(nitrogen_levels):
    print(f"Sample {index + 1}: Soil Nitrogen Level = {level} mg/kg")
```

```
Sample 1: Soil Nitrogen Level = 15 mg/kg
Sample 2: Soil Nitrogen Level = 20 mg/kg
Sample 3: Soil Nitrogen Level = 10 mg/kg
Sample 4: Soil Nitrogen Level = 25 mg/kg
Sample 5: Soil Nitrogen Level = 18 mg/kg
```

In this example, `index` represents the position of each element in the list (starting from 0), and `level` is the nitrogen level. We use `index + 1` in the `print` statement to start the sample numbering from 1 instead of 0.

## 22.4 Example 3: Combine for loop with if statement

Combining a `for` loop with `if` statements unleashes a powerful and precise control over data processing and decision-making within iterative sequences. The `for` loop provides a structured way to iterate over a range of elements in a collection, such as lists, tuples, or strings. When an `if` statement is nested within this loop, it introduces conditional logic, allowing the program to execute specific blocks of code only when certain criteria are met. This combination is incredibly versatile: it can be used for filtering data, conditional aggregation of data, and applying different operations to elements based on specific conditions.

In this short example we will combine a `for` loop with `if` statements to generate the complementary DNA strand by iterating over each nucleotide. The code will also filter if there is an incorrect base and in which position that incorrect base is located.

```
# Example of DNA strand
strand = 'ACCTTATCGGC'

# Create an empty complementary strand
strand_c = ''
```

```

# Iterate over each base in the DNA strand (a string)
for k,base in enumerate(strand):

    if base == 'A':
        strand_c += 'T'

    elif base == 'T':
        strand_c += 'A'

    elif base == 'C':
        strand_c += 'G'

    elif base == 'G':
        strand_c += 'C'

    else:
        print('Incorrect base', base, 'in position', k+1)

print(strand_c)

```

TGGAATAGCCG

Try to insert or change one of the bases in the sequence for another character not representing a DNA nucleotide.

## 22.5 Example 4: For loop using a dictionary

```

# Record air temperatures for a few cities in Kansas
kansas_weather = {
    "Topeka": {"Record High Temperature": 40, "Date": "July 20, 2003"},
    "Wichita": {"Record High Temperature": 42, "Date": "August 8, 2010"},
    "Lawrence": {"Record High Temperature": 39, "Date": "June 15, 2006"},
    "Manhattan": {"Record High Temperature": 41, "Date": "July 18, 2003"}
}

# Iterating through the dictionary
for city, weather_details in kansas_weather.items():
    print(f"Record Weather in {city}:")
    print(f"  High Temperature: {weather_details['Record High Temperature']}°C")

```

```

print(f" Date of Occurrence: {weather_details['Date']}")

Record Weather in Topeka:
    High Temperature: 40°C
    Date of Occurrence: July 20, 2003
Record Weather in Wichita:
    High Temperature: 42°C
    Date of Occurrence: August 8, 2010
Record Weather in Lawrence:
    High Temperature: 39°C
    Date of Occurrence: June 15, 2006
Record Weather in Manhattan:
    High Temperature: 41°C
    Date of Occurrence: July 18, 2003

```

## 22.6 Example 5: Nested for loops

Imagine we are analyzing soil samples from different fields. Each field has multiple samples, and each sample has various measurements. We'll use nested for loops to iterate through the fields and then through each measurement in the samples.

```

# Soil data from multiple fields
soil_data = {
    "Field 1": [
        {"pH": 6.5, "Moisture": 20, "Nitrogen": 3},
        {"pH": 6.8, "Moisture": 22, "Nitrogen": 3.2}
    ],
    "Field 2": [
        {"pH": 7.0, "Moisture": 18, "Nitrogen": 2.8},
        {"pH": 7.1, "Moisture": 19, "Nitrogen": 2.9}
    ]
}

# Iterating through each field
for field, samples in soil_data.items():
    print(f"Data for {field}:")

    # Nested loop to iterate through each sample in the field
    for sample in samples:

```

```

print(f" Sample - pH: {sample['pH']}, Moisture: {sample['Moisture']}%, Nitrogen: {sample['Nitrogen']}%")

Data for Field 1:
    Sample - pH: 6.5, Moisture: 20%, Nitrogen: 3%
    Sample - pH: 6.8, Moisture: 22%, Nitrogen: 3.2%
Data for Field 2:
    Sample - pH: 7.0, Moisture: 18%, Nitrogen: 2.8%
    Sample - pH: 7.1, Moisture: 19%, Nitrogen: 2.9%

```

In this example, `soil_data` is a dictionary where each key is a field, and the value is a list of soil samples (each sample is a dictionary of measurements). The first for loop iterates over the fields, and the nested loop iterates over the samples within each field, printing out the `pH`, `Moisture`, and `Nitrogen` content for each sample.

## 22.7 Example 6: For loop using break and continue

Imagine we are evaluating crop yields from different fields. We want to stop processing if we encounter a field with exceptionally low yield (signifying a possible data error or a major issue with the field) and skip over fields with average yields to focus on fields with exceptionally high or low yields.

```

# Crop yield data (in tons per hectare) for different fields
crop_yields = {"Field 1": 2.5, "Field 2": 3.2, "Field 3": 1.0, "Field 4": 3.8, "Field 5": 2.0}

# Thresholds for yield consideration
low_yield_threshold = 1.5
high_yield_threshold = 3.0

for field, yield_data in crop_yields.items():
    if (yield_data < low_yield_threshold) or (yield_data > high_yield_threshold):
        print(f"{field} is a potential outlier: {yield_data} tons/ha")
        break # Stop processing further as this could indicate a major issue
    else:
        continue

```

Field 2 is a potential outlier: 3.2 tons/ha

We use `break` to stop the iteration when we encounter a yield below the `low_yield_threshold` or above `high_yield_threshold`, which could indicate an outlier that requires immediate attention.

We use `continue` to skip to the next iteration without executing any additional code in the loop.

## 22.8 Compatve anatomy of for loops

### 22.8.0.0.1 Python

```
nitrogen_levels = [15, 20, 10, 25, 18]
for level in nitrogen_levels:
    print(f"Soil Nitrogen Level: {level} mg/kg")
```

### 22.8.0.0.2 Matlab

```
nitrogen_levels = [15, 20, 10, 25, 18];
for level = nitrogen_levels
    fprintf('Soil Nitrogen Level: %d mg/kg\n', level);
end
```

### 22.8.0.0.3 Julia

```
nitrogen_levels = [15, 20, 10, 25, 18]
for level in nitrogen_levels
    println("Soil Nitrogen Level: $level mg/kg")
end
```

### 22.8.0.0.4 R

```
nitrogen_levels <- c(15, 20, 10, 25, 18)
for (level in nitrogen_levels) {
  cat("Soil Nitrogen Level:", level, "mg/kg\n")}
```

### 22.8.0.0.5 JavaScript

```
const nitrogen_levels = [15, 20, 10, 25, 18];
for (let level of nitrogen_levels) {
    console.log(`Soil Nitrogen Level: ${level} mg/kg`);
}
```

#### 22.8.0.0.6 Commonalities among programming languages:

- All languages use a `for` keyword to start the loop.
- They iterate over a collection of items, like an array or list.
- Each language uses a variable to represent the current item in each iteration.
- The body of the loop (code to be executed) is enclosed within a block defined by indentation or brackets.

# 23 While loop

A `while` loop is used to repeatedly execute a block of code as long as a certain condition remains true. Unlike `for` loops, the `while` loop is particularly useful when the number of iterations isn't predetermined, and you need to keep running the code until a specific condition is met or changes.

## 23.1 Syntax

```
while condition:  
    # Code to execute repeatedly
```

`condition`: A Boolean expression that determines whether the loop continues.

## 23.2 Example 1: A trivial loop

This example will only print values 0, 1, and 2. As soon as the third iteration ends, the value of `A` becomes 3 and the while loop breaks when attempting to start the fourth iteration since the condition `A < 3` is no longer met.

```
A = 0  
while A < 3:  
    print(A)  
    A += 1
```

```
0  
1  
2
```

### 23.3 Example 2: Guess the soil taxonomic order

The soil taxonomic orders form the highest category in the soil classification system, each order representing distinct characteristics and soil formation processes. The twelve orders provide a framework for understanding soil properties and their implications for agriculture, environmental management, and land use. [This website](#) has great pictures of real soil profiles and additional useful information.

It's always fun to play games, but crafting your own is even better. This exercise was written so that you can change the information contained in the `database` dictionary with some other data that sparks your interest. For soil scientists, this is an excellent opportunity to review your knowledge about the soil taxonomy.

```
import random

# Dictionary of soil orders with a hint
database = {
    "Alfisols": ["Fertile, with subsurface clay accumulation.", "Starts with 'A'"],
    "Andisols": ["Formed from volcanic materials, high in organic matter.", "Starts with 'A'"],
    "Aridisols": ["Dry soils, often found in deserts.", "Starts with 'A'"],
    "Entisols": ["Young soils with minimal horizon development.", "Starts with 'E'"],
    "Gelisols": ["Contain permafrost, found in cold regions.", "Starts with 'G'"],
    "Histosols": ["Organic, often water-saturated soils like peat.", "Starts with 'H'"],
    "Inceptisols": ["Young, with weak horizon development.", "Starts with 'I'"],
    "Mollisols": ["Dark, rich in organic matter, found under grasslands.", "Starts with 'M'"],
    "Oxisols": ["Highly weathered, tropical soils.", "Starts with 'O'"],
    "Spodosols": ["Acidic, with organic and mineral layers.", "Starts with 'S'"],
    "Ultisols": ["Weathered, acidic soils with subsurface clay.", "Starts with 'U'"],
    "Vertisols": ["Rich in clay, expand and contract with moisture.", "Starts with 'V'"]
}

# Select a random soil taxonomic order and its hints
selection, hints = random.choice(list(database.items()))
hints_iter = iter(hints)

print("Guess the soil taxonomic order! Type 'hint' for a hint.")

# Initial hint
print("First Hint:", next(hints_iter))

# While loop for guessing game
```

```

while True:
    guess = input("Your guess: ").strip().lower()

    if guess == selection.lower():
        print(f"Correct! It was {selection}.")
        break

    elif guess == "hint":
        try:
            print("Hint:", next(hints_iter))
        except StopIteration:
            print("No more hints. Please make a guess.")
    else:
        print("Incorrect, try again or type 'hint' for another hint.")

```

Guess the soil taxonomic order! Type 'hint' for a hint.  
First Hint: Highly weathered, tropical soils.

Your guess: Oxisols

Correct! It was Oxisols.

### 23.3.1 Explanation

The `iter()` function is used to create an iterator from an iterable object, like a list or a dictionary. Once you have an iterator, you can use the `next()` function to sequentially access elements from the iterator. Each call to `next()` retrieves the next element in the sequence. When `next()` reaches the end of the sequence and there are no more elements to return, it raises a `StopIteration` exception. This combination allows for a controlled iteration process, especially useful in situations where you need to process elements one at a time.

The `strip()` method removes any leading and trailing whitespace (like spaces, tabs, or new lines) from the input string. This is helpful for ensuring that extra spaces do not affect the comparison of the user's guess to the correct answer.

The `lower()` method then converts the string to lowercase. This ensures that the comparison is case-insensitive, meaning that “Dandelion”, “dandelion”, and “DANDELION” are all treated as the same guess.

## **23.4 References**

- The twelve orders of soil taxonomy. United States Department of Agriculture website: <https://www.nrcs.usda.gov/resources/education-and-teaching-materials/the-twelve-orders-of-soil-taxonomy>. Accessed on 8 January 2024

# 24 Objects and classes

Python's programming approach is centered around objects, a concept that encapsulates data and functionality. In Python, everything you use —be it variables, modules, or figures— is an object. This approach is part of what we call object-oriented programming (OOP), which is useful because it organizes code into small, reusable pieces, making it easier to understand and maintain. Think of objects as mini-programs that have their own properties (like characteristics or data) and methods (functions or actions they can perform).

In Python, objects often have both properties and methods. A property is like an attribute of an object, while a method is a function that belongs to the object and usually performs some action or computation related to the object. Methods are called with parentheses and may take arguments.

## 24.1 Syntax

Syntax for accessing object properties and methods:

```
object.property  
object.method()
```

Syntax for defining our own objects using classes:

```
class ClassName:  
    def __init__(self, attributes):  
        # Constructor method for initializing instance attributes  
  
    def method_name(self, parameters):  
        # Method of the class
```

`ClassName` is the name of the class. `__init__` is the constructor method. `method_name` is a method of the class.

Let's look at these concepts using a simple example and then we will create our own object.

## 24.2 Properties and methods example

I find that using the NumPy module is often a simple and clear example new for students to grasp the difference between properties/attributes and methods/functions.

### Jargon note

The terms **property** and **attribute** are used interchangeably to represent characteristics of the object.

Similarly, the term **method** is another word to denote a **function** inside of an object. Methods (or functions) represent actions that can be performed on the object and are only available within a specific object.

```
import numpy as np

# Define an array (this is an object)
A = np.array([1, 2, 3, 4, 5, 6])

# Properties of the Numpy array object
print('Properties')
print(A.shape) # Dimensions of the array
print(A.size) # Total number of elements
print(A.dtype) # Data type
print(A.ndim) # Number of array dimensions

# Methods/Functions of the Numpy array object
print('') # Add a blank line
print('Methods')
print(A.mean()) # Method to compute average of all values
print(A.sum()) # Method to compute sum of all values
print(A.cumsum()) # Method to compute running sum of all values
print(A.reshape(3,2)) # Reshape to a 3 by 2 matrix
```

Properties

(6,)

6

int64

1

Methods

3.5

```
21
[ 1  3  6 10 15 21]
[[1 2]
 [3 4]
 [5 6]]
```

## 24.3 Class example: Laboratory sample

Consider a scenario where we operate a soil analysis laboratory receiving samples from various clients, such as farmers, gardeners, and golf course superintendents. Each soil sample possesses unique attributes that we need to record and analyze. These attributes might include the client's full name, the date the sample was received, a unique identifier, the location of sample collection, the analyses requested, and the results of these analyses. In this context, the primary unit of interest is the individual sample, and all the additional information represents its metadata.

To efficiently manage this data, we can create a Python class specifically for our soil samples. This class will allow us to create a structured record for each new sample we receive, demonstrating Python's flexibility in creating custom objects tailored to our specific needs. We will use Python's `uuid` module to generate a unique identifier for each sample and the `datetime` module to timestamp when each sample object is created.

 Note

Classes are like blueprints for objects since they define what properties and methods an object will have. For more information check Python's official documentation about [objects and classes](#)

```
import uuid
import datetime
import pprint

class SoilSample:
    """ Class that defines attributes and methods for new soil samples"""

    def __init__(self, customer_name, location, analyses_requested):
        """Attributes of the soil sample generated upon sample entry.

        Inputs
        -----
        customer_name : string
```

```

        Customer's full name
location : tuple
    Geographic location (lat,lon) of sample collection
analyses_requested : list
    Requested analyses
"""

self.sample_id = str(uuid.uuid4()) # Unique identifier for each sample
self.timestamp = datetime.datetime.now().strftime("%d-%b-%Y %H:%M:%S") # Timestamp
self.customer_name = customer_name # Customer's full name
self.location = location # Geographic location of sample collection
self.analyses_requested = analyses_requested # List of requested analyses
self.results = {} # Dictionary to store results of analyses

def add_results(self, analysis_type, result):
    """Function that adds the name and results of a specific soil analysis."""
    self.results[analysis_type] = result # Add analysis results

def summary(self):
    """Function that prints summary information for the sample."""
    info = (f"Sample ID: {self.sample_id}",
            f"Timestamp: {self.timestamp}",
            f"Customer: {self.customer_name}",
            f"Location: {self.location}",
            f"Requested Analyses: {self.analyses_requested}",
            f"Results: {self.results}")
    return pprint.pprint(info)

```

**i** What is `__init__()`?

The `__init__()` function at the beginning of the class is a special method that gets called (another term for this action is *invoked*) automatically when we create a new instance of the class. Think of `__init__` as the setup of the object, where the initial state of a new object is defined by assigning values to its properties.

**i** What is `self`?

When defining a class, the word `self` is used to refer to the instance of the class itself. It's a way for the class to reference its own attributes and methods and is usually defined with the words `self`, `this`, or anything else you want. Typically is a short word that is easy to type. We will use `self` to match the official Python documentation.

Imagine each class as a blueprint for building a house. Each house built from the blueprint is an instance (an occurrence) of the class. In this context, `self` is like saying *this particular house*, rather than the general blueprint.

```
# Access our own documentation
SoilSample?
```

```
Init signature: SoilSample(customer_name, location, analyses_requested)
Docstring:      <no docstring>
Init docstring:
Attributes of the soil sample generated upon sample entry.
```

Inputs

-----

```
customer_name : string
    Customer's full name
location : tuple
    Geographic location (lat,lon) of sample collection
analyses_requested : list
    Requested analyses
Type:           type
Subclasses:
```

```
# Example usage. Create or instantiate a new object, in this case a new sample.
new_sample = SoilSample("Andres Patrignani", (39.210089, -96.590213), ["pH", "Nitrogen"])
```

### **i** What does *instantiation* mean?

Instantiation is the term used in Python for the process of creating a new object from a blueprint, the class we just defined.

```
# Access properties generated when we created the new sample
print(new_sample.customer_name)
print(new_sample.timestamp)
```

```
Andres Patrignani
04-Jan-2024 16:52:03
```

```

# Use the add_results() method to add some information to our sample object
new_sample.add_results("pH", 6.5)
new_sample.add_results("Nitrogen", 20)

# Use the summary() method to print the information available for our sample
new_sample.summary()

('Sample ID: 322a5b20-ed71-4c4e-b181-789fb6574d8d',
 'Timestamp: 04-Jan-2024 16:52:03',
 'Customer: Andres Patrignani',
 'Location: (39.210089, -96.590213)',
 "Requested Analyses: ['pH', 'Nitrogen']",
 "Results: {'pH': 6.5, 'Nitrogen': 20}")

```

**i** Classes are powerful

A cool feature of classes in Python is their ability to inherit properties and methods from an already pre-defined class. This is called **inheritance**, and it allows programmers to build upon and extend the functionality of existing classes, creating new, more specialized versions without reinventing the wheel. After mastering the basics of classes and objects, exploring class inheritance is a must to take your coding skills to the next level.

## 24.4 Practice

To take this exercise to the next level try the following improvements:

- Add one new attribute and one new methods to the class
- Use the `input()` function to request the required data for each sample from users
- Use a `for` loop to pre-populate the results with `None` for each of the requested soil analyses. This will ensure that only those analyses are entered into the sample.
- Create a way to store multiple sample entries. You can simply append each new sample to a variable defined at the beginning of your script, append the new entries to a text or `.json` file, use the `pickle` module, or use databases like `sqlite3`, `MySQL`, or `TinyDB` module

# 25 Error handling

Error handling is a technique for managing and responding to exceptions, which are errors detected during execution of code. Effective error handling ensures that your program can deal with unexpected situations without crashing. This is particularly important in data-driven fields, where the input data might be unpredictable or in an unexpected format.

## 25.1 Syntax

The primary constructs for error handling are `try`, `except`, `else`, and `finally`:

`try`: This block lets you test a block of code for errors.

`except`: This block handles the error or specifies a response to specific error types.

`else`: (Optional) This block runs if no errors are raised in the try block.

`finally`: (Optional) This block executes regardless of the result of the try-except blocks.

```
try:  
    # Code block where exceptions might occur  
except SomeException:  
    # Code to handle the exception  
else:  
    # Code to execute if no exceptions occur (optional)  
finally:  
    # Code to execute regardless of exceptions (optional)
```

In addition to catching exceptions, Python allows programmers to raise their own exceptions using the `raise` statement. This can be useful for signaling specific error types in a way that is clear and tailored to your program's needs.

Python also has the `isinstance()` function that enables easy checking of strict data types. This is a Pythonic and handy method to validate input data types in functions.

## 25.2 Example: Classification of soil acidity-alkalinity

To illustrate these error handling concepts, let's write a simple function to determine whether a soil is acidic or alkaline based on its soil pH. This function will not only require that we pass a numeric input to the function, but also that the range of soil pH is within 0 and 14.

```
def test_soil_pH(pH_value):
    """
    Determines if soil is acidic, alkaline, or neutral based on its pH value.

    Parameters:
    pH_value (int, float, or str): The pH value of the soil. It should be a number
                                    or a string that can be converted to a float.
                                    Valid pH values range from 0 to 14.

    Returns:
    str: A description of the soil's acidity or alkalinity.

    Raises:
    TypeError: If the input is not an int, float, or string.
    ValueError: If the input is not within the valid pH range (0 to 14).
    """

    if not isinstance(pH_value, (int, float, str)):
        raise TypeError(f"Input type {type(pH_value)} is not valid. Must be int, float, or str")

    try:
        pH_value = float(pH_value)
        if not (0 <= pH_value <= 14):
            raise ValueError("pH value must be between 0 and 14.")
    except ValueError as e:
        return f"Invalid input: {e}"

    # Classify the soil based on its pH value
    if pH_value < 7:
        return "The soil is acidic."
    elif pH_value > 7:
        return "The soil is alkaline."
    else:
        return "The soil is neutral."
```

```
# Example usage of the function
print(test_soil_pH("5.5")) # Acidic soil
print(test_soil_pH(8.2)) # Alkaline soil
print(test_soil_pH("seven")) # Invalid input
print(test_soil_pH(15)) # Valid input, out of range
print(test_soil_pH([7.5])) # Invalid type
```

The soil is acidic.

The soil is alkaline.

Invalid input: could not convert string to float: 'seven'

Invalid input: pH value must be between 0 and 14.

TypeError: Input type <class 'list'> is not valid. Must be int, float, or str.

## 25.3 Explanation

**Data type check with `isinstance`:** At the beginning of the function, we use `isinstance()` to check if the input is either an int, float, or str. The `TypeError` message includes the type of the wrong data type to inform the user about the nature of the error.

**Conversion and range check:** Inside the `try` block, the function attempts to convert the input to a float and then checks if it's within the valid pH range, raising a `ValueError` if it's out of range.

**Handling value error:** The `except` block catches and returns a message for any `ValueError`.

# 26 Numpy module

Numpy is a Python library for efficient numerical computing that is widely used in data science and engineering. Numpy provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level functions to operate on these arrays. The true power of Numpy lies in the ability to vectorize computations, which allow for element-wise operations on arrays without the need for explicit loops. This feature enhances performance and efficiency, particularly in data-intensive tasks. [Numpy Documentation](#)

## 26.1 Element-wise computations using Numpy arrays

We will start this tutorial by learning about some of the limitations of traditional Python lists. Then, the power of Numpy element-wise (or vectorized) computations will become evident. As much as I like agronomic examples, for the first few exercises I will use some trivial arrays of numbers to keep it simple.

Vectorized or element-wise computations refer to operations that are performed on arrays (vectors, matrices) directly and simultaneously. Instead of processing elements one by one using loops, vectorized operations apply a single action to each element in the array without explicit iteration. This leads to more concise code and often improved performance. Both `vectorized` and `element-wise` terms are correct and often are used interchangeably.

Let's begin by importing the Numpy module, so that we already have it for the entire tutorial.

```
# Import numpy module
import numpy as np
import matplotlib.pyplot as plt # We will need this for some figures
```

### 26.1.1 Product of a regular list by a scalar

```
# Create a list of elements
A = [1,2,3,4]
```

```
# Multiple the list by a scalar
print(A * 3)
```

```
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

### 26.1.2 Product of two regular lists with the same shape

```
# Create list B, with the same size as A
B = [5,6,7,8]

# Multiple the two lists together. Heads up! This will not work!
print(A*B)
```

```
TypeError: can't multiply sequence by non-int of type 'list'
```

The first operation repeats the list three times, which is probably not exactly what you were expecting. The second example results in an error. Now let's import the Numpy module and try these operations again to see what happens.

### 26.1.3 Product of a Numpy array by a scalar

```
# Create a list of elements
A = np.array([1,2,3,4])

# Check type of newly created Numpy array
print(type(A))

print(A * 3)

<class 'numpy.ndarray'>
[ 3  6  9 12]
```

#### 26.1.4 Product of two Numpy arrays with the same shape

```
# Re-define the previous arrays as numpy arrays
A = np.array([1,2,3,4])
B = np.array([5,6,7,8])

print(A * B)
```

```
[ 5 12 21 32]
```

#### 26.1.5 Other operations with Numpy arrays

```
print(A * 3)      # Vector times a scalar
print(A + B)
print(A - B)
print(A * B)
print(A / B)
print(np.sqrt(A**2 + B**2)) # Exponentiation Calculate hypotenuse of multiple rectangle triangles
print(A.sum())
print(B.sum())
```

```
[ 3   6   9  12]
[ 6   8  10  12]
[-4  -4  -4  -4]
[ 5 12 21 32]
[0.2          0.33333333 0.42857143 0.5          ]
[5.09901951  6.32455532  7.61577311  8.94427191]
10
26
```

## 26.2 Example 1: Compute soil water storage for a single field

Soil water storage directly influences plant growth and crop yield since it is an essential processes for transpiration and nutrient uptake. In irrigated cropping systems, monitoring of soil water storage is important for irrigation scheduling and effective water management and conservation.

Assume a field that has a soil profile with five horizons, each measuring: 10, 20, 30, 30, and 30 cm in thickness (so about 1.2 m depth). The volumetric water content for each horizon

was determined by soil moisture sensors located at the center of each horizon, hence providing an average moisture value of: 0.350, 0.280, 0.255, 0.210, 0.137  $cm^3/cm^3$ . Based on this information, compute the soil water storage for each soil horizon and the total for the entire soil profile. Recall that the volumetric water content represents the volume of water per unit volume of soil, so 0.350  $cm^3/cm^3$  is the same as 35% moisture by volume or by thickness of the soil horizon. How much irrigation water does a farmer need to add to reach a field capacity of 420 mm?

```
# Define variables
theta_v = np.array([0.350, 0.280, 0.255, 0.210, 0.137]) # cm^3/cm^3
depths = np.array([10, 20, 30, 30, 30]) * 10 # horizons in mm

# Compute water storage for each layer
storage_per_layer = theta_v * depths # mm of water per layer
print('Storage for each layer:', np.round(storage_per_layer,1))

# Compute soil water storage for the entire profile
profile_storage = np.sum(storage_per_layer)
print(f'Total water storage in profile is: {profile_storage:.1f} mm')

field_capacity = 420 # mm
irrigation_req = field_capacity - profile_storage
print(f'Required irrigation: {irrigation_req:.1f} mm')
```

```
Storage for each layer: [35. 56. 76.5 63. 41.1]
Total water storage in profile is: 271.6 mm
Required irrigation: 148.4 mm
```

### Note

Let's review a few operations:

```
depths = np.array([10, 20, 30, 30, 30]) * 10 is the product of a vector and a scalar
storage_per_layer = theta_v * depths is a vector times another vector
```

## 26.3 Example 2: Compute soil water storage for multiple fields

Now imagine that we are managing three irrigated fields in the region? Assume that all the fields are nearby and have the same soil horizons, but farmers have different crops and irrigation strategies so they also have different soil moisture contents across the profile. What

is the soil water storage of each field? How much water do we need to apply in each field to bring them back to field capacity?

```
# Example soil moisture for the 5 horizons and three fields
# The first field is the same as in the previous exercise
theta_v = np.array([[0.350, 0.280, 0.255, 0.210, 0.137],
                    [0.250, 0.380, 0.355, 0.110, 0.250],
                    [0.150, 0.180, 0.155, 0.110, 0.320]]) # cm^3/cm^3

# Compute storage for all horizons
storage_per_layer = theta_v * depths
print(storage_per_layer)

# Compute
storage_profiles = np.sum(storage, axis=1) # axis=1 means add along columns
print(storage_profiles)

# Irrigation requirements
irrigation_req = field_capacity - storage_profiles
print('Irrigation for each field:', irrigation_req, 'mm')

[[ 35.   56.   76.5  63.   41.1]
 [ 25.   76.  106.5  33.   75. ]
 [ 15.   36.   46.5  33.   96. ]]
[271.6 315.5 226.5]
Irrigation for each field: [148.4 104.5 193.5] mm
```

### 26.3.1 Example 3: Determine the CEC of a soil

The cation exchange capacity (CEC, meq/100 g of soil) of a soil is determined by the nature and amount of clay minerals and organic matter. Compute the CEC of a soil that has 32% clay and 3% organic matter. The clay fraction is represented by 30% kaolinite, 50% montmorillonite, and 20% vermiculite. The CEC for clay minerals and organic matter can be found in most soil fertility textbooks.

```
# Determine percentage of each clay mineral

om = np.array([4]) # percent
om_cec = np.array([200]) # meq/100 g

clay = 32 * np.array([30, 50, 20])/100 # This is the % of each clay type
```

```
clay_cec = np.array([10, 100, 140]) # meq/100 g

# Merge the fractions and CEC together into a single array
all_fractions = np.concatenate((om, clay))/100 # percent to fraction
all_cec = np.concatenate((om_cec, clay_cec))

print(all_fractions)
print(all_cec)
```

```
[0.04 0.096 0.16 0.064]
[200 10 100 140]
```

```
# Compute soil CEC as the weighed-sum of its components
soil_cec = np.sum(all_cec * all_fractions)
print(f'The soil CEC is: {soil_cec:.1f} meq/100 g of soil')
```

```
The soil CEC is: 33.9 meq/100 g of soil
```

#### i Note

Let's review a few operations:

`np.array([30, 50, 20])/100` is a vector divided by a scalar.  
`all_cec * all_fractions` is a vector times another vector

### 26.3.2 Create arrays with specific data types

```
# An alternative by specifying the data type
print(np.array([1,2,3,4], dtype="int64"))
print(np.array([1,2,3,4], dtype="float64"))
```

```
[1 2 3 4]
[1. 2. 3. 4.]
```

#### i Note

A common pitfall among beginners is to create a numpy array only using parentheses like this: `array = np.array(1,2,3,4)`. This will not work.

### 26.3.3 Operations with two-dimensional arrays

```
# Define arrays
# The values in M and v were arbitrarily selected
# so that the operations result in round numbers for clarity. You can change them.

M = np.array([ [10,2,1], [25,6,55] ]) # 2D matrix
v = np.array([0.2, 0.5, 1]) # 1D vector

# Access Numpy array shape and size properties
print(M.shape) # rows and columns
print(M.size) # Total number of elements

(2, 3)
6

# Element-wise multiplication
print('Matrix by a scalar')
print(M*2)

print('Matrix by a vector with same number of columns')
print(M * v)

print('Matrix by matrix of the same size')
print(M*M)

Matrix by a scalar
[[ 20   4    2]
 [ 50   12   110]]
Matrix by a vector with same number of columns
[[ 2.   1.   1.]
 [ 5.   3.  55.]]
Matrix by matrix of the same size
[[ 100     4      1]
 [ 625    36  3025]]

# Dot product (useful for linear algebra operations)
print('Dot product operation')
np.dot(M,v)
```

```
Dot product operation
```

```
array([ 4., 63.])
```

#### 26.3.4 Reshape arrays

```
# Reshape M array (original 2 rows 3 columns)
print(M)

# Reshape to 3 rows and 2 columns
print(M.reshape(3,2))

# Reshape to 1 row and 0 columns (1D array)
print(M.reshape(6,1))
print(M.reshape(6,1).shape) # Check the shape

# Similar
```

```
[[10  2  1]
 [25  6 55]]
[[10  2]
 [ 1 25]
 [ 6 55]]
[[10]
 [ 2]
 [ 1]
 [25]
 [ 6]
 [55]]
(6, 1)
```

#### 26.4 Numpy boolean operations

```
expr1 = np.array([1,2,3,4]) == 3
print(expr1)

expr2 = np.array([1,2,3,4]) == 2
print(expr2)
```

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

# Elements in both vectors need to match to be considered True
print(expr1 & expr2) # print(expr1 and expr2)

# It is sufficient with a single match in one of the vectors
print(expr1 | expr2) # print(expr1 or expr2)

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

## 26.5 Flattening

Sometimes we want to serialize our 2D or 3D matrix into a long one-dimensional vector. This operation is called “flattening” and Numpy has a specific method to carry this operation that is called *flattening* and array.

```
# Flatten two-dimensional array M
M.flatten()

array([10,  2,  1, 25,  6, 55])
```

## 26.6 Use the Numpy random module to create a random image

To show some of the power of working with Numpy arrays we will create a random image. Images in the RGB (red-green-blue) color space are represented by three matrices that encode the color of each pixel. Colors are represented with an integer between 0 and 255. This is called an 8-bit integer, and there are only  $2^8 = 256$  possible integers. Because each image has three bands (one for red, one for green, and one for blue) there is a total of 17 million ( $256 \times 256 \times 256$  or  $256^3$ ) possible colors for each pixel.

```
# Define image size (keep it small so that you can see each pixel)
rows = 20
cols = 30

# Set seed for reproducibility.
# Everyone will obtain the same random numbers
```

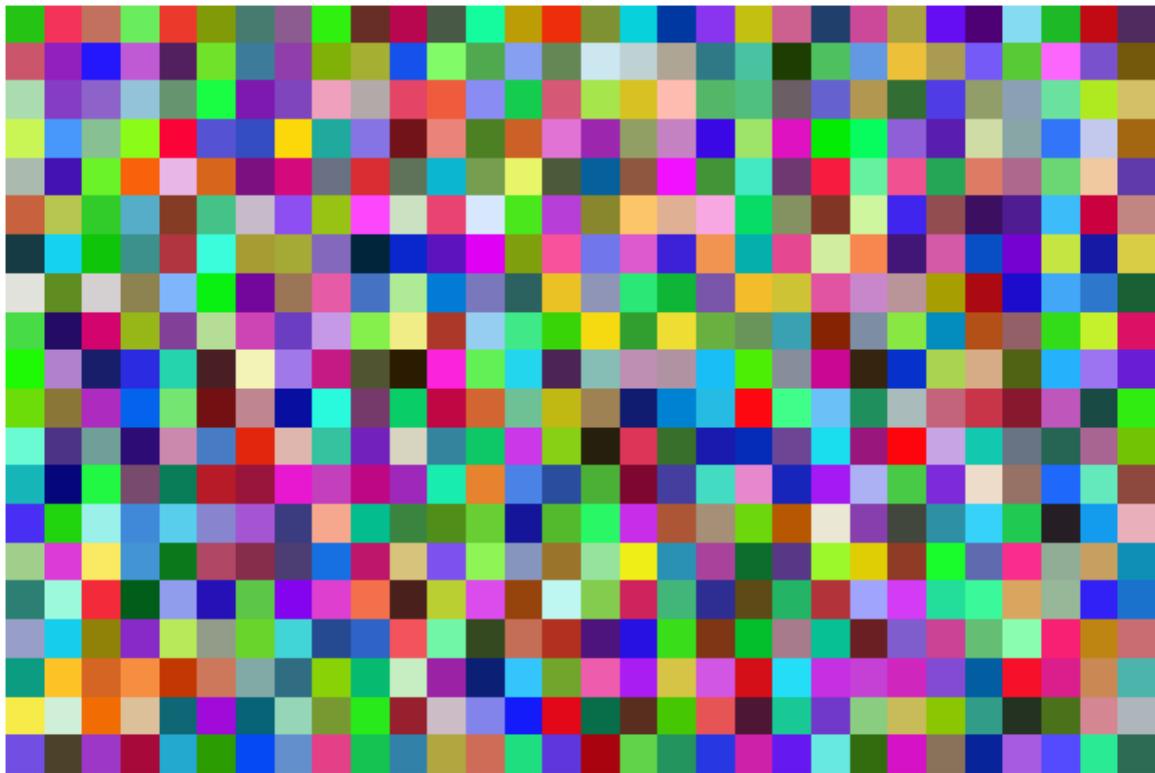
```
np.random.seed(1)

# Create image bands
# uint8 means unsigned integer of 8-bits
R = np.random.randint(0, 255, [rows,cols], dtype='uint8')
G = np.random.randint(0, 255, [rows,cols], dtype='uint8')
B = np.random.randint(0, 255, [rows,cols], dtype='uint8')

# Stack image bands (axis=2 means along the third dimension, or on top of each other)
RGB = np.stack( (R,G,B), axis=2)
print('Image size:', RGB.shape) # Shape of the RGB variable

# Display image using the matplotlib library (we imported this at the top)
plt.figure(figsize=(8,8))
plt.imshow(RGB)
plt.axis('off') # Mute this line to see what the image looks like without it.
plt.show()
```

Image size: (20, 30, 3)



## 26.7 Numpy handy functions

```
# Generate a range of integers
# np.arange(start,stop,step)
print('range()')
print(np.arange(0,100,10))

# Generate linear range
# numpy.linspace(start, stop, num=50, endpoint=True)
print('')
print('linspace()')
print(np.linspace(0, 10, 5))

# Array of zeros
print('')
print('zeros()')
print(np.zeros([5,3]))
```

```

# Array of ones
print('')
print('ones()')
print(np.ones([4,3]))

# Array of NaN values
print('')
print('full()')
print(np.full([4,3], np.nan)) # This also works
print(np.ones([4,3])*np.nan)

# Meshgrid (first create 1D vectors, then create a 2D mesh)
N = 5
lat = np.linspace(36, 40, N)
lon = np.linspace(-102, -98, N)
LAT,LON = np.meshgrid(lat,lon)
print('')
print('Grid of latitudes')
print(LAT)
print('')
print('Grid of longitudes')
print(LON)

range()
[ 0 10 20 30 40 50 60 70 80 90]

linspace()
[ 0. 2.5 5. 7.5 10. ]

zeros()
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]

ones()
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]

```

```

full()
[[nan nan nan]
 [nan nan nan]
 [nan nan nan]
 [nan nan nan]]

Grid of latitudes
[[36. 37. 38. 39. 40.]
 [36. 37. 38. 39. 40.]
 [36. 37. 38. 39. 40.]
 [36. 37. 38. 39. 40.]
 [36. 37. 38. 39. 40.]]

Grid of longitudes
[[-102. -102. -102. -102. -102.]
 [-101. -101. -101. -101. -101.]
 [-100. -100. -100. -100. -100.]
 [-99. -99. -99. -99. -99.]
 [-98. -98. -98. -98. -98.]]

```

## 26.8 Create a noisy wave

With Numpy we can easily implement models, create timeseries, add noise, and perform trigonometric operations. In this example we will create a synthetic timeseries of air temperature using a cosine wave. To make this more realistic we will also add some noise.

```

# Set random seed for reproducibility
np.random.seed(1)

# Define wave inputs
T_avg = 15 # Annual average in Celsius
A = 10 # Annual amplitude [Celsius]
doy = np.arange(1,366) # Vector of days of the year

# Generate x and y axis
x = 2 * np.pi * doy/365 # Convert doy into pi-radians
y = T_avg - A*np.cos(x) # Sine wave

# Add random noise
noise = np.random.normal(0, 3, x.size) # White noise having zero mean

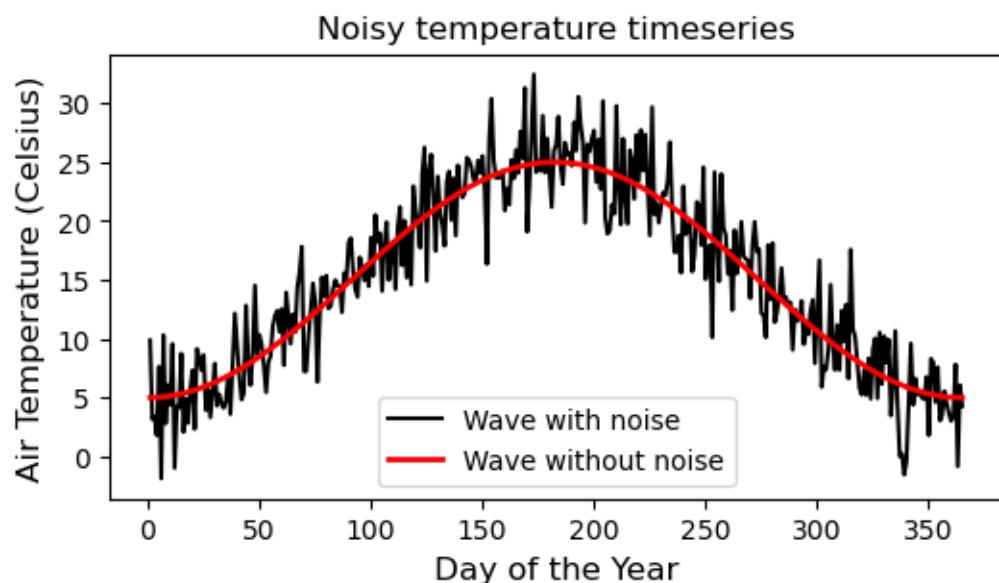
```

```

y_noisy = y + noise

# Visualize wave using Matplotlib
plt.figure(figsize=(6,3))
plt.title('Noisy temperature timeseries')
plt.plot(doy, y_noisy, '-k', label="Wave with noise")
plt.plot(doy, y, '-r', linewidth=2, label="Wave without noise")
plt.xlabel('Day of the Year', size=12)
plt.ylabel('Air Temperature (Celsius)', size=12)
plt.legend()
plt.show()

```



### 26.8.1 Descriptive stats

To finish our tutorial, let's inspect Numpy methods to obtain some descriptive statistics for the wave we created earlier.

```

# Descriptive stats
print('Mean:', y.mean()) # Arithmetic average
print('Standard deviation:', y.std()) # Standard deviation
print('Variance:', y.var()) # Variance
print('Median:', np.median(y)) # Median
print('Minimum:', y.min()) # Minimum

```

```
print('Maximum:', y.max()) # Maximum
print('Index of minimum:', y.argmin()) # Position of minimum value
print('Index of maximum:', y.argmax()) # Position of maximum value
print('50th percentile:', np.percentile(y, 50)) # 50th percentile (should equal to the med)
print('5th and 95th percentiles:', np.percentile(y, [5,95])) # 5th and 95th percentile
```

```
Mean: 15.0
Standard deviation: 7.0710678118654755
Variance: 50.0
Median: 15.000000000000007
Minimum: 5.000092602638098
Maximum: 24.999907397361902
Index of minimum: 273
Index of maximum: 90
50th percentile: 15.000000000000007
5th and 95th percentiles: [ 5.12930816 24.87069184]
```

## 26.9 Reference

Walt, S.V.D., Colbert, S.C. and Varoquaux, G., 2011. The NumPy array: a structure for efficient numerical computation. Computing in science & engineering, 13(2), pp.22-30.

# 27 Pandas module

Pandas is a powerful library primarily used for tabular data analysis. It offers data structures like DataFrames and Series, which make tasks such as data cleaning and aggregation easy. Pandas integrates well with other libraries like Numpy, and can read data saved in various text formats and spreadsheet software like MS Excel. One of the hallmarks of Pandas Dataframes is that we can call data stored in rows and columns by name, similar to working with Python dictionaries.

```
# Import modules
import pandas as pd
import numpy as np
```

## 27.1 Create DataFrame from existing variable

After importing the module we have two possible directions. We import data from a file or we convert an existing variable into a Pandas DataFrame. Here we will create a simple DataFrame to learn the basics. This way we will be able to display the result of our operations without worrying about extensive datasets.

Let's create a dictionary with some weather data and missing values (represented by -9999).

```
# Create dictionary with some weather data
data = {'timestamp': ['1/1/2000', '2/1/2000', '3/1/2000', '4/1/2000', '5/1/2000'],
        'windSpeed': [2.2, 3.2, -9999.0, 4.1, 2.9],
        'windDirection': ['E', 'NW', 'NW', 'N', 'S'],
        'precipitation': [0, 18, 25, 2, 0]}
```

The next step consists of converting the dictionary into a Pandas DataFrame. This is straight forward using the DataFrame method of the Pandas module pd.DataFrame()

```
# Convert dictionary into DataFrame
df = pd.DataFrame(data)
df.head()
```

|   | timestamp | windSpeed | windDirection | precipitation |
|---|-----------|-----------|---------------|---------------|
| 0 | 1/1/2000  | 2.2       | E             | 0             |
| 1 | 2/1/2000  | 3.2       | NW            | 18            |
| 2 | 3/1/2000  | -9999.0   | NW            | 25            |
| 3 | 4/1/2000  | 4.1       | N             | 2             |
| 4 | 5/1/2000  | 2.9       | S             | 0             |

To display the DataFrame content simply use the `head()` and `tail()` methods. As an alternative you can use the `print()` function or type the name of the DataFrame and hit `ctrl + Enter`. Note that by default Jupyter Lab highlights the different table rows when using the second option, so for readability purposes I will use the second option from now on.

Dissecting the DataFrame above we find the following main components:

1. header row containing column names
2. index (the left-most column with numbers from 0 to 4) is equivalent to a row name.
3. Each column has data of the same type.

```
# By default, values in Pandas series are Numpy arrays
print(df["windSpeed"].values)
```

```
print(type(df["windSpeed"].values))
```

```
[ 2.200e+00  3.200e+00 -9.999e+03  4.100e+00  2.900e+00]
<class 'numpy.ndarray'>
```

## 27.2 Basic methods and properties

Pandas DataFrame has dedicated functions to display a limited number of heading and tailing rows.

```
df.head(3) # First three rows
```

|   | timestamp | windSpeed | windDirection | precipitation |
|---|-----------|-----------|---------------|---------------|
| 0 | 1/1/2000  | 2.2       | E             | 0             |
| 1 | 2/1/2000  | 3.2       | NW            | 18            |
| 2 | 3/1/2000  | -9999.0   | NW            | 25            |

```
df.tail(3) # Last three rows
```

|   | timestamp | windSpeed | windDirection | precipitation |
|---|-----------|-----------|---------------|---------------|
| 2 | 3/1/2000  | -9999.0   | NW            | 25            |
| 3 | 4/1/2000  | 4.1       | N             | 2             |
| 4 | 5/1/2000  | 2.9       | S             | 0             |

To start handling our data we need to learn how to retrieve data from the Pandas DataFrame. If we don't know the column names of the dataset we can print them using the column property. We can also inspect the data type of each column as well as its total number of elements and shape.

```
df.columns # Column names
```

```
Index(['timestamp', 'windSpeed', 'windDirection', 'precipitation'], dtype='object')
```

```
df.size # Total number of elements
```

```
20
```

```
df.shape # Number of rows and columns
```

```
(5, 4)
```

```
df.dtypes # Data type for each column
```

```
timestamp      object
windSpeed     float64
windDirection   object
precipitation    int64
dtype: object
```

## 27.3 Convert strings to datetime

```

# Convert dates in string format to Pandas datetime format
# %d = day in format 00 days
# %m = month in format 00 months
# %Y = full year

df["timestamp"] = pd.to_datetime(df["timestamp"], format="%d/%m/%Y")
df.head()

```

|   | timestamp  | windSpeed | windDirection | precipitation |
|---|------------|-----------|---------------|---------------|
| 0 | 2000-01-01 | 2.2       | E             | 0             |
| 1 | 2000-01-02 | 3.2       | NW            | 18            |
| 2 | 2000-01-03 | -9999.0   | NW            | 25            |
| 3 | 2000-01-04 | 4.1       | N             | 2             |
| 4 | 2000-01-05 | 2.9       | S             | 0             |

```

# Note that the format of our `timestamp` column change to datetime format
df.dtypes

```

```

timestamp      datetime64[ns]
windSpeed       float64
windDirection    object
precipitation    int64
dtype: object

```

## 27.4 Extract information from the timestamp

Here we can obtain the day, month, year, and even other components such hours, minutes and nanoseconds from the timestamp. Having a separate column for some of these components can be extremely helpful in case we want to aggregate data. For instance, to compute the monthly mean air temperature we need to know in what month each temperature record was obtained.

For this we will use the `dt` submodule within Pandas.

```

# Get the day of the year
df["doy"] = df["timestamp"].dt.dayofyear
df.head()

```

|   | timestamp  | windSpeed | windDirection | precipitation | doy |
|---|------------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 2.2       | E             | 0             | 1   |
| 1 | 2000-01-02 | 3.2       | NW            | 18            | 2   |
| 2 | 2000-01-03 | -9999.0   | NW            | 25            | 3   |
| 3 | 2000-01-04 | 4.1       | N             | 2             | 4   |
| 4 | 2000-01-05 | 2.9       | S             | 0             | 5   |

Note that the new column was placed at the end. This is the default when creating a new column.

The next example makes use of the `insert()` method to add the new column on a specific location. Typically for dates and date components we want to have the columns at the beginning, close to the datetime. For other variables the previous approach that appends the new column at the end of the DataFrame will work just fine.

```
# Get month from timestamp and create new column
#.insert(positionOfNewColumn, nameOfNewColumn, dataOfNewColumn)

df.insert(1,'month',df["timestamp"].dt.month)
df.head()
```

|   | timestamp  | month | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 1     | 2.2       | E             | 0             | 1   |
| 1 | 2000-01-02 | 1     | 3.2       | NW            | 18            | 2   |
| 2 | 2000-01-03 | 1     | -9999.0   | NW            | 25            | 3   |
| 3 | 2000-01-04 | 1     | 4.1       | N             | 2             | 4   |
| 4 | 2000-01-05 | 1     | 2.9       | S             | 0             | 5   |

If you re-run the previous cell you will get an error since Pandas prevents having two columns with the same name.

## 27.5 Missing values

One of the most common operations when working with data is to handle missing values. Almost every dataset has missing data and there is no universal way of denoting missing values. Most common placeholders are: `NaN`, `NA`, `-99`, `-9999`, `missing`. To find out more about how missing data is represented in your dataset read associated documentation files.

I typically prefer to use `NaN` (personal preference since I also use Matlab frequently). In many datasets missing data may already be imported as `NaN` or `NA`, meaning that you can directly use the `fillna()` method without this intermediary step.

To replace missing values we will follow these steps:

1. Identify the cells with `-9999` values. Output will be a logical DataFrame having the same dimensions as `df`.
2. Replace `-9999` with `NaN` values. Note that I'm using `NaN` values from **Numpy**.
3. Check our work using the `isna()` method (optional)

Pandas offers a machinery to deal with missing data, meaning that it is not necessary to replace these values in order to make computations with the data. We just need to ensure that missing values are in the right format. In some cases an option would be to replace missing data with an estimate value, like using the average of the values immediately above and below or using the average for the entire data (not ideal) using the `fillna()` method. For instance:  
`df.fillna(df.windSpeed.mean())`

```
# Step 1: find -9999 values across the entire dataframe
```

```
idx_missing = df.isin([-9999])
idx_missing
```

|   | timestamp | month | windSpeed | windDirection | precipitation | doy   |
|---|-----------|-------|-----------|---------------|---------------|-------|
| 0 | False     | False | False     | False         | False         | False |
| 1 | False     | False | False     | False         | False         | False |
| 2 | False     | False | True      | False         | False         | False |
| 3 | False     | False | False     | False         | False         | False |
| 4 | False     | False | False     | False         | False         | False |

```
# Find missing values in only one column
df["windSpeed"] == -9999
```

```
0    False
1    False
2     True
3    False
4    False
Name: windSpeed, dtype: bool
```

```
# Step 2: Replace missing values with NaN
```

```
df[idx_missing] = np.nan  
df
```

|   | timestamp  | month | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 1     | 2.2       | E             | 0             | 1   |
| 1 | 2000-01-02 | 1     | 3.2       | NW            | 18            | 2   |
| 2 | 2000-01-03 | 1     | NaN       | NW            | 25            | 3   |
| 3 | 2000-01-04 | 1     | 4.1       | N             | 2             | 4   |
| 4 | 2000-01-05 | 1     | 2.9       | S             | 0             | 5   |

```
# NaNs are of type float  
type(np.nan)
```

```
float
```

```
# Step 3: Check our work  
df.isna()
```

|   | timestamp | month | windSpeed | windDirection | precipitation | doy   |
|---|-----------|-------|-----------|---------------|---------------|-------|
| 0 | False     | False | False     | False         | False         | False |
| 1 | False     | False | False     | False         | False         | False |
| 2 | False     | False | True      | False         | False         | False |
| 3 | False     | False | False     | False         | False         | False |
| 4 | False     | False | False     | False         | False         | False |

## 27.6 Quick statistics

DataFrames have a variety of methods to calculate simple statistics. To obtain an overall summary we can use the `describe()` method.

```
# Summary stats for all columns  
print(df.describe())
```

```

      month  windSpeed  precipitation      doy
count      5.0    4.000000      5.000000  5.000000
mean       1.0    3.100000      9.000000  3.000000
std        0.0    0.787401     11.7047   1.581139
min        1.0    2.200000      0.000000  1.000000
25%        1.0    2.725000      0.000000  2.000000
50%        1.0    3.050000      2.000000  3.000000
75%        1.0    3.425000     18.000000  4.000000
max        1.0    4.100000     25.000000  5.000000

# Metric ignoring NaN values
print(df["windSpeed"].max())          # Maximum value for each column
print(df["windSpeed"].mean())          # Average value for each column
print(df["windSpeed"].min())          # Minimum value for each column
print(df["windSpeed"].std())           # Standard deviation value for each column
print(df["windSpeed"].var())           # Variance value for each column
print(df["windSpeed"].median())         # Variance value for each column
print(df["windSpeed"].quantile(0.95))

4.1
3.1
2.2
0.7874007874011809
0.6199999999999997
3.05
3.964999999999994

# Cumulative sum. Useful to compute cumulative precipitation
print(df.precipitation.cumsum())

0      0
1     18
2     43
3     45
4     45
Name: precipitation, dtype: int64

# Unique values. Useful to compute unique wind directions
print(df.windDirection.unique())

['E' 'NW' 'N' 'S']

```

## 27.7 Indexing and slicing

### 27.7.1 Using index operator []

To start making computations with need to access the data inside the Pandas DataFrame. Indexing and slicing are useful operations to select portions of data by calling specific rows, columns, or a combination of both. The index operator [] is primarily intended to be used with column labels (e.g. `df[columnName]`), however, it can also handle row slices (e.g. `df[rows]`). A common notation useful to understand how the slicing works is as follows:

## 27.8 Select rows

```
df[0:3] # First three rows
```

|   | timestamp  | month | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 1     | 2.2       | E             | 0             | 1   |
| 1 | 2000-01-02 | 1     | 3.2       | NW            | 18            | 2   |
| 2 | 2000-01-03 | 1     | NaN       | NW            | 25            | 3   |

## 27.9 Select columns

We can call individual columns using the dot or bracket notation. Note that in option 2 there is no . between df and `['windSpeed']`

```
df.windSpeed      # Option 1  
df['windSpeed'] # Option 2
```

```
0    2.2  
1    3.2  
2    NaN  
3    4.1  
4    2.9  
Name: windSpeed, dtype: float64
```

To pass more than one row of column you will need to group them in a list.

```
# Select multiple columns at once
df[['windSpeed','windDirection']]
```

|   | windSpeed | windDirection |
|---|-----------|---------------|
| 0 | 2.2       | E             |
| 1 | 3.2       | NW            |
| 2 | NaN       | NW            |
| 3 | 4.1       | N             |
| 4 | 2.9       | S             |

A common mistake when slicing multiple columns is to forget grouping column names into a list, so the following command will not work:

```
df['windSpeed','windDirection']
```

## 27.10 Slicing rows and columns

### 27.10.1 Using iloc method

**iloc:** Integer-location. iloc gets rows (or columns) at specific indexes. It only takes integers as input. **Exclusive of its endpoint**

```
# Top 3 rows and columns 1 and 2
df.iloc[0:3,[1,2]]
```

|   | month | windSpeed |
|---|-------|-----------|
| 0 | 1     | 2.2       |
| 1 | 1     | 3.2       |
| 2 | 1     | NaN       |

```
# Top 2 rows and all columns
df.iloc[0:2,:] # Same as: df.iloc[0:2]
```

|   | timestamp  | month | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 1     | 2.2       | E             | 0             | 1   |
| 1 | 2000-01-02 | 1     | 3.2       | NW            | 18            | 2   |

Although a bit more verbose and perhaps less *pythonic*, I prefer to specify `all the columns` using the `:` character. In my opinion this notation is more explicit and clearly states the rows and columns of the slicing operation. So, `df.iloc[0:2,:]` is preferred over `df.iloc[0:2]`.

### 27.10.2 Using `loc` method

`loc` gets rows (or columns) with specific labels. **Inclusive of its endpoint**

```
# Select multiple rows and columns at once using the loc method
df.loc[0:2,['windSpeed','windDirection']]
```

|   | windSpeed | windDirection |
|---|-----------|---------------|
| 0 | 2.2       | E             |
| 1 | 3.2       | NW            |
| 2 | NaN       | NW            |

```
# Some rows and all columns
df.loc[0:1,:]
```

|   | timestamp  | month | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 1     | 2.2       | E             | 0             | 1   |
| 1 | 2000-01-02 | 1     | 3.2       | NW            | 18            | 2   |

```
# First three elements of a single column
df.loc[0:2,'windSpeed']
```

```
0    2.2
1    3.2
2    NaN
Name: windSpeed, dtype: float64
```

```
# First three elements of multiple columns
df.loc[0:2,['windSpeed','windDirection']]
```

|   | windSpeed | windDirection |
|---|-----------|---------------|
| 0 | 2.2       | E             |
| 1 | 3.2       | NW            |
| 2 | NaN       | NW            |

These statements will not work with `loc`:

```
df.loc[0:2,0:1]
df.loc[[0:2],[0:1]]
```

## 27.11 Filter data using boolean indexing

Boolean indexing (a.k.a. logical indexing) consists of creating a boolean array with True/False values as a consequence of conditional statement that can be used to select the rows that meet the specified condition. Logical indexing are often not intuitive for students learning how to program, but are incredibly powerful and I highly encourage its use.

Let's select all the data for days that have wind speed greater than 3 meters per second. We will first select the rows of `df.windSpeed` that are greater than 3 m/s, and then we will use the resulting boolean to slice the DataFrame.

```
idx = df.windSpeed > 3 # Rows in which the wind speed is greater than
idx # Let's inspect the idx variable.
```

```
0    False
1     True
2    False
3     True
4    False
Name: windSpeed, dtype: bool
```

```
# Now let's apply the boolean variable to the dataframe
df[idx]
```

|   | timestamp  | month | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|-----------|---------------|---------------|-----|
| 1 | 2000-01-02 | 1     | 3.2       | NW            | 18            | 2   |
| 3 | 2000-01-04 | 1     | 4.1       | N             | 2             | 4   |

```
# We can also apply the boolean to specific columns  
df.loc[idx,'windDirection']
```

```
1    NW  
3     N  
Name: windDirection, dtype: object
```

It's also possible to write the previous command with the conditional statement in a single line of code. This is fine to do, but sometimes nesting too many conditions can create commands that are hard to understand. I typically recommend storing the boolean results in a new variable that is easier to pass around. This is particularly helpful if you are planning to re-use the boolean in multiple lines of your code.

```
# Same in a single line of code  
df.loc[df.windSpeed > 3,'windDirection']
```

```
1    NW  
3     N  
Name: windDirection, dtype: object
```

Another popular way of filtering is to check whether an element or group of elements are within a set. If you come from Matlab the following example is similar to the `ismember()` function.

Let's check whether January 1 and January 2 are in the dataframe.

```
idx_doy = df['doy'].isin([1,2]) #list(range(1,5))  
idx_doy
```

```
0    True  
1    True  
2   False  
3   False  
4   False  
Name: doy, dtype: bool
```

```
df.loc[idx_doy,:] # Select all columns for the selected days of the year
```

|   | timestamp  | month | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 1     | 2.2       | E             | 0             | 1   |
| 1 | 2000-01-02 | 1     | 3.2       | NW            | 18            | 2   |

## 27.12 Pandas custom date range

In this particular case we have the day of the year to indicate time, however, in many occasions is better to handle dates. Often times dates are already present in the dataset, but if they aren't then we can create and handle dates with Pandas.

```
subset_dates = pd.date_range('20000102', periods=2, freq='D') # Used df.shape[0] to find t
subset_dates

DatetimeIndex(['2000-01-02', '2000-01-03'], dtype='datetime64[ns]', freq='D')

# The same to generate months
pd.date_range('20200101', periods=df.shape[0], freq='M') # Specify the frequency to months

DatetimeIndex(['2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',
               '2020-05-31'],
              dtype='datetime64[ns]', freq='M')
```

## 27.13 Select range of dates with boolean indexing

Now that we covered both boolean indexing and pandas dates we can use these concepts to select data from a specific window of time. This is a pretty common operation when trying to select a subset of the entire DataFrame by a specific date range.

```
# Generate boolean for rows that match the subset of dates generated earlier
idx_subset = df["timestamp"].isin(subset_dates)
idx_subset

0    False
1     True
2     True
3    False
```

```
4    False
```

```
Name: timestamp, dtype: bool
```

```
# Generate a new DataFrame using only the rows with matching dates
df_subset = df.loc[idx_subset]
df_subset
```

|   | timestamp  | month | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|-----------|---------------|---------------|-----|
| 1 | 2000-01-02 | 1     | 3.2       | NW            | 18            | 2   |
| 2 | 2000-01-03 | 1     | NaN       | NW            | 25            | 3   |

```
# It isn't always necessary to generate a new DataFrame
# So you can access a specific column like this
df.loc[idx_subset, "precipitation"]
```

```
1    18
2    25
```

```
Name: precipitation, dtype: int64
```

## 27.14 Add and remove columns

The `insert()` and `drop()` methods allow us to add or remove columns to/from the DataFrame. The most common use of these functions is as follows:

```
df.insert(indexOfNewColumn, nameOfNewColumn, dataArrayOfNewColumn)
```

```
df.drop(nameOfColumnToBeRemoved)
```

```
# Add new column at a specific location
```

```
df.insert(2, 'airTemperature', [25.4, 26, 27.1, 28.9, 30.2]) # Similar to: df['dates'] = df
```

|   | timestamp  | month | airTemperature | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|----------------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 1     | 25.4           | 2.2       | E             | 0             | 1   |
| 1 | 2000-01-02 | 1     | 26.0           | 3.2       | NW            | 18            | 2   |
| 2 | 2000-01-03 | 1     | 27.1           | NaN       | NW            | 25            | 3   |
| 3 | 2000-01-04 | 1     | 28.9           | 4.1       | N             | 2             | 4   |
| 4 | 2000-01-05 | 1     | 30.2           | 2.9       | S             | 0             | 5   |

```
timestamp month airTemperature windSpeed windDirection precipitation doy
```

```
# Remove specific column  
df.drop(columns=['airTemperature'])
```

|   | timestamp  | month | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 1     | 2.2       | E             | 0             | 1   |
| 1 | 2000-01-02 | 1     | 3.2       | NW            | 18            | 2   |
| 2 | 2000-01-03 | 1     | NaN       | NW            | 25            | 3   |
| 3 | 2000-01-04 | 1     | 4.1       | N             | 2             | 4   |
| 4 | 2000-01-05 | 1     | 2.9       | S             | 0             | 5   |

## 27.15 Reset DataFrame index

```
# Replace the index by a variables of our choice  
df.set_index('doy')
```

| doy | timestamp  | month | airTemperature | windSpeed | windDirection | precipitation |
|-----|------------|-------|----------------|-----------|---------------|---------------|
| 1   | 2000-01-01 | 1     | 25.4           | 2.2       | E             | 0             |
| 2   | 2000-01-02 | 1     | 26.0           | 3.2       | NW            | 18            |
| 3   | 2000-01-03 | 1     | 27.1           | NaN       | NW            | 25            |
| 4   | 2000-01-04 | 1     | 28.9           | 4.1       | N             | 2             |
| 5   | 2000-01-05 | 1     | 30.2           | 2.9       | S             | 0             |

```
# Reset the index (see that 'doy' goes back to the end of the DataFrame again)  
df.reset_index(0)
```

|   | index | timestamp  | month | airTemperature | windSpeed | windDirection | precipitation | doy |
|---|-------|------------|-------|----------------|-----------|---------------|---------------|-----|
| 0 | 0     | 2000-01-01 | 1     | 25.4           | 2.2       | E             | 0             | 1   |
| 1 | 1     | 2000-01-02 | 1     | 26.0           | 3.2       | NW            | 18            | 2   |
| 2 | 2     | 2000-01-03 | 1     | 27.1           | NaN       | NW            | 25            | 3   |
| 3 | 3     | 2000-01-04 | 1     | 28.9           | 4.1       | N             | 2             | 4   |
| 4 | 4     | 2000-01-05 | 1     | 30.2           | 2.9       | S             | 0             | 5   |

## 27.16 Merge two dataframes

```
# Create a new DataFrame (follows dates)

# Dictionary
data2 = {'timestamp': ['6/1/2000', '7/1/2000', '8/1/2000', '9/1/2000', '10/1/2000'],
         'windSpeed': [4.3, 2.1, 0.5, 2.7, 1.9],
         'windDirection': ['N', 'N', 'SW', 'E', 'NW'],
         'precipitation': [0, 0, 0, 25, 0]}

# Dictionary to DataFrame
df2 = pd.DataFrame(data2)

# Convert strings to pandas datetime
df2["timestamp"] = pd.to_datetime(df2["timestamp"], format="%d/%m/%Y")

df2.head()
```

|   | timestamp  | windSpeed | windDirection | precipitation |
|---|------------|-----------|---------------|---------------|
| 0 | 2000-01-06 | 4.3       | N             | 0             |
| 1 | 2000-01-07 | 2.1       | N             | 0             |
| 2 | 2000-01-08 | 0.5       | SW            | 0             |
| 3 | 2000-01-09 | 2.7       | E             | 25            |
| 4 | 2000-01-10 | 1.9       | NW            | 0             |

Not using the `format="%d/%m/%y"` in the previous cell results in the wrong datetime conversion. It is always recommended to specify the format.

```
# Merge both Dataframes using an union of keys from both frames (how='outer' option)
df_merged = pd.merge(df, df2, how='outer')
df_merged
```

|   | timestamp  | month | airTemperature | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|----------------|-----------|---------------|---------------|-----|
| 0 | 2000-01-01 | 1.0   | 25.4           | 2.2       | E             | 0             | 1.0 |
| 1 | 2000-01-02 | 1.0   | 26.0           | 3.2       | NW            | 18            | 2.0 |
| 2 | 2000-01-03 | 1.0   | 27.1           | NaN       | NW            | 25            | 3.0 |
| 3 | 2000-01-04 | 1.0   | 28.9           | 4.1       | N             | 2             | 4.0 |
| 4 | 2000-01-05 | 1.0   | 30.2           | 2.9       | S             | 0             | 5.0 |
| 5 | 2000-01-06 | NaN   | NaN            | 4.3       | N             | 0             | NaN |

|   | timestamp  | month | airTemperature | windSpeed | windDirection | precipitation | doy |
|---|------------|-------|----------------|-----------|---------------|---------------|-----|
| 6 | 2000-01-07 | NaN   | NaN            | 2.1       | N             | 0             | NaN |
| 7 | 2000-01-08 | NaN   | NaN            | 0.5       | SW            | 0             | NaN |
| 8 | 2000-01-09 | NaN   | NaN            | 2.7       | E             | 25            | NaN |
| 9 | 2000-01-10 | NaN   | NaN            | 1.9       | NW            | 0             | NaN |

Note how NaN values were assigned to variables not present in the new DataFrame

```
# Create another DataFrame with more limited data. Values every other day
data3 = {'timestamp': ['1/1/2000', '3/1/2000', '5/1/2000', '7/1/2000', '9/1/2000'],
         'pressure': [980, 987, 985, 991, 990]} # Pressure in millibars
df3 = pd.DataFrame(data3)
df3["timestamp"] = pd.to_datetime(df3["timestamp"], format="%d/%m/%Y")
df3.head()
```

|   | timestamp  | pressure |
|---|------------|----------|
| 0 | 2000-01-01 | 980      |
| 1 | 2000-01-03 | 987      |
| 2 | 2000-01-05 | 985      |
| 3 | 2000-01-07 | 991      |
| 4 | 2000-01-09 | 990      |

```
# Only the matching rows will be merged
df_merged.merge(df3, on="timestamp")
```

|   | timestamp  | month | airTemperature | windSpeed | windDirection | precipitation | doy | pressure |
|---|------------|-------|----------------|-----------|---------------|---------------|-----|----------|
| 0 | 2000-01-01 | 1.0   | 25.4           | 2.2       | E             | 0             | 1.0 | 980      |
| 1 | 2000-01-03 | 1.0   | 27.1           | NaN       | NW            | 25            | 3.0 | 987      |
| 2 | 2000-01-05 | 1.0   | 30.2           | 2.9       | S             | 0             | 5.0 | 985      |
| 3 | 2000-01-07 | NaN   | NaN            | 2.1       | N             | 0             | NaN | 991      |
| 4 | 2000-01-09 | NaN   | NaN            | 2.7       | E             | 25            | NaN | 990      |

```
# Only add values from the new, more sporadic, variable where there is a match.
df_merged.merge(df3, how="left")
```

|   | timestamp  | month | airTemperature | windSpeed | windDirection | precipitation | doy | pressure |
|---|------------|-------|----------------|-----------|---------------|---------------|-----|----------|
| 0 | 2000-01-01 | 1.0   | 25.4           | 2.2       | E             | 0             | 1.0 | 980.0    |
| 1 | 2000-01-02 | 1.0   | 26.0           | 3.2       | NW            | 18            | 2.0 | NaN      |
| 2 | 2000-01-03 | 1.0   | 27.1           | NaN       | NW            | 25            | 3.0 | 987.0    |
| 3 | 2000-01-04 | 1.0   | 28.9           | 4.1       | N             | 2             | 4.0 | NaN      |
| 4 | 2000-01-05 | 1.0   | 30.2           | 2.9       | S             | 0             | 5.0 | 985.0    |
| 5 | 2000-01-06 | NaN   | NaN            | 4.3       | N             | 0             | NaN | NaN      |
| 6 | 2000-01-07 | NaN   | NaN            | 2.1       | N             | 0             | NaN | 991.0    |
| 7 | 2000-01-08 | NaN   | NaN            | 0.5       | SW            | 0             | NaN | NaN      |
| 8 | 2000-01-09 | NaN   | NaN            | 2.7       | E             | 25            | NaN | 990.0    |
| 9 | 2000-01-10 | NaN   | NaN            | 1.9       | NW            | 0             | NaN | NaN      |

## 27.17 Operations with real dataset

```
# Read CSV file
data_url = "https://raw.githubusercontent.com/soilwater/pynotes-agriscience/gh-pages/datas
df = pd.read_csv(data_url, sep=',')
df.head(5)
```

|   | STID | NAME    | ST | LAT   | LON    | YR   | MO | DA | HR | MI | ... | RELH | CHIL | HEAT | WDI |
|---|------|---------|----|-------|--------|------|----|----|----|----|-----|------|------|------|-----|
| 0 | ACME | Acme    | OK | 34.81 | -98.02 | 2019 | 4  | 15 | 15 | 20 | ... |      |      |      |     |
| 1 | ADAX | Ada     | OK | 34.80 | -96.67 | 2019 | 4  | 15 | 15 | 20 | ... | 40   |      |      | S   |
| 2 | ALTU | Altus   | OK | 34.59 | -99.34 | 2019 | 4  | 15 | 15 | 20 | ... | 39   |      | 82   | SSW |
| 3 | ALV2 | Alva    | OK | 36.71 | -98.71 | 2019 | 4  | 15 | 15 | 20 | ... | 32   |      | 82   | S   |
| 4 | ANT2 | Antlers | OK | 34.25 | -95.67 | 2019 | 4  | 15 | 15 | 20 | ... | 35   |      |      | S   |

Some columns seem to have empty cells. Ideally we would like to see these cells filled with `NaN` values. Something looks fishy here. Let's inspect some of these cell.

```
# Print one the cells to see what's in there
df.loc[0, 'RAIN']
```

```
1
```

In the inspected cell we found a string with a single space in it. Now we can use the `replace()` method to substitute these strings for `NaN` from the Numpy module.

The `inplace=True` replaces the string with `NaN` without generating a copy of the Pandas Dataframe

```
df.replace(' ', np.nan, inplace=True)  
df.head(5)
```

|   | STID | NAME    | ST | LAT   | LON    | YR   | MO | DA | HR | MI | ... | RELH | CHIL | HEAT | WDI |
|---|------|---------|----|-------|--------|------|----|----|----|----|-----|------|------|------|-----|
| 0 | ACME | Acme    | OK | 34.81 | -98.02 | 2019 | 4  | 15 | 15 | 20 | ... | NaN  | NaN  | NaN  | NaN |
| 1 | ADAX | Ada     | OK | 34.80 | -96.67 | 2019 | 4  | 15 | 15 | 20 | ... | 40   | NaN  | NaN  | S   |
| 2 | ALTU | Altus   | OK | 34.59 | -99.34 | 2019 | 4  | 15 | 15 | 20 | ... | 39   | NaN  | 82   | SSW |
| 3 | ALV2 | Alva    | OK | 36.71 | -98.71 | 2019 | 4  | 15 | 15 | 20 | ... | 32   | NaN  | 82   | S   |
| 4 | ANT2 | Antlers | OK | 34.25 | -95.67 | 2019 | 4  | 15 | 15 | 20 | ... | 35   | NaN  | NaN  | S   |

### 💡 Tip

The previous solution is not the best. We could have resolved the issue with the empty strings by simply adding the following option `na_values=' '` to the `pd.read_csv()` function, like this: `df = pd.read_csv(data_url, sep=',', na_values=' ')`. This will automatically populate all cells with `' '` with `NaN` values. We just did not know this in advance.

#### 27.17.1 Match specific stations

```
idx_acme = df['STID'].str.match('ACME')  
df[idx_acme]
```

|   | STID | NAME | ST | LAT   | LON    | YR   | MO | DA | HR | MI | ... | RELH | CHIL | HEAT | WDI |
|---|------|------|----|-------|--------|------|----|----|----|----|-----|------|------|------|-----|
| 0 | ACME | Acme | OK | 34.81 | -98.02 | 2019 | 4  | 15 | 15 | 20 | ... | NaN  | NaN  | NaN  | NaN |

```
idx_starts_with_A = df['STID'].str.match('A')  
df[idx_starts_with_A]
```

|   | STID | NAME  | ST | LAT   | LON    | YR   | MO | DA | HR | MI | ... | RELH | CHIL | HEAT | WDI |
|---|------|-------|----|-------|--------|------|----|----|----|----|-----|------|------|------|-----|
| 0 | ACME | Acme  | OK | 34.81 | -98.02 | 2019 | 4  | 15 | 15 | 20 | ... | NaN  | NaN  | NaN  | NaN |
| 1 | ADAX | Ada   | OK | 34.80 | -96.67 | 2019 | 4  | 15 | 15 | 20 | ... | 40   | NaN  | NaN  | S   |
| 2 | ALTU | Altus | OK | 34.59 | -99.34 | 2019 | 4  | 15 | 15 | 20 | ... | 39   | NaN  | 82   | SSW |

|   | STID | NAME    | ST | LAT   | LON    | YR   | MO | DA | HR | MI | ... | RELH | CHIL | HEAT | WI |
|---|------|---------|----|-------|--------|------|----|----|----|----|-----|------|------|------|----|
| 3 | ALV2 | Alva    | OK | 36.71 | -98.71 | 2019 | 4  | 15 | 15 | 20 | ... | 32   | NaN  | 82   | S  |
| 4 | ANT2 | Antlers | OK | 34.25 | -95.67 | 2019 | 4  | 15 | 15 | 20 | ... | 35   | NaN  | NaN  | S  |
| 5 | APAC | Apache  | OK | 34.91 | -98.29 | 2019 | 4  | 15 | 15 | 20 | ... | 41   | NaN  | NaN  | S  |
| 6 | ARD2 | Ardmore | OK | 34.19 | -97.09 | 2019 | 4  | 15 | 15 | 20 | ... | 41   | NaN  | NaN  | S  |
| 7 | ARNE | Arnett  | OK | 36.07 | -99.90 | 2019 | 4  | 15 | 15 | 20 | ... | 10   | NaN  | 85   | SW |

```
idx_has_A = df['STID'].str.contains('A')
df[idx_has_A].head(15)
```

|    | STID | NAME                | ST | LAT   | LON     | YR   | MO | DA | HR | MI | ... | RELH | CHIL | HEAT | WI |
|----|------|---------------------|----|-------|---------|------|----|----|----|----|-----|------|------|------|----|
| 0  | ACME | Acme                | OK | 34.81 | -98.02  | 2019 | 4  | 15 | 15 | 20 | ... | NaN  | NaN  |      |    |
| 1  | ADAX | Ada                 | OK | 34.80 | -96.67  | 2019 | 4  | 15 | 15 | 20 | ... | 40   | NaN  |      |    |
| 2  | ALTU | Altus               | OK | 34.59 | -99.34  | 2019 | 4  | 15 | 15 | 20 | ... | 39   | NaN  |      |    |
| 3  | ALV2 | Alva                | OK | 36.71 | -98.71  | 2019 | 4  | 15 | 15 | 20 | ... | 32   | NaN  |      |    |
| 4  | ANT2 | Antlers             | OK | 34.25 | -95.67  | 2019 | 4  | 15 | 15 | 20 | ... | 35   | NaN  |      |    |
| 5  | APAC | Apache              | OK | 34.91 | -98.29  | 2019 | 4  | 15 | 15 | 20 | ... | 41   | NaN  |      |    |
| 6  | ARD2 | Ardmore             | OK | 34.19 | -97.09  | 2019 | 4  | 15 | 15 | 20 | ... | 41   | NaN  |      |    |
| 7  | ARNE | Arnett              | OK | 36.07 | -99.90  | 2019 | 4  | 15 | 15 | 20 | ... | 10   | NaN  |      |    |
| 8  | BEAV | Beaver              | OK | 36.80 | -100.53 | 2019 | 4  | 15 | 15 | 20 | ... | 9    | NaN  |      |    |
| 11 | BLAC | Blackwell           | OK | 36.75 | -97.25  | 2019 | 4  | 15 | 15 | 20 | ... | 38   | NaN  |      |    |
| 20 | BYAR | Byars               | OK | 34.85 | -97.00  | 2019 | 4  | 15 | 15 | 20 | ... | 43   | NaN  |      |    |
| 21 | CAMA | Camargo             | OK | 36.03 | -99.35  | 2019 | 4  | 15 | 15 | 20 | ... | 32   | NaN  |      |    |
| 22 | CARL | Lake Carl Blackwell | OK | 36.15 | -97.29  | 2019 | 4  | 15 | 15 | 20 | ... | 36   | NaN  |      |    |
| 24 | CHAN | Chandler            | OK | 35.65 | -96.80  | 2019 | 4  | 15 | 15 | 20 | ... | 37   | NaN  |      |    |
| 28 | CLAY | Clayton             | OK | 34.66 | -95.33  | 2019 | 4  | 15 | 15 | 20 | ... | 36   | NaN  |      |    |

```
idx = df['NAME'].str.contains('Blackwell') & df['NAME'].str.contains('Lake')
df[idx]
```

|    | STID | NAME                | ST | LAT   | LON    | YR   | MO | DA | HR | MI | ... | RELH | CHIL | HEAT | WI |
|----|------|---------------------|----|-------|--------|------|----|----|----|----|-----|------|------|------|----|
| 22 | CARL | Lake Carl Blackwell | OK | 36.15 | -97.29 | 2019 | 4  | 15 | 15 | 20 | ... | 36   | NaN  |      |    |

The following line won't work because the string matching is case sensitive:

```
idx = df['NAME'].str.contains('LAKE')
```

```
idx = df['NAME'].str.contains('Blackwell') | df['NAME'].str.contains('Lake')
df[idx]
```

|    | STID | NAME                | ST | LAT   | LON    | YR   | MO | DA | HR | MI | ... | RELH | CHIL | HEAT | WDD |
|----|------|---------------------|----|-------|--------|------|----|----|----|----|-----|------|------|------|-----|
| 11 | BLAC | Blackwell           | OK | 36.75 | -97.25 | 2019 | 4  | 15 | 15 | 20 | ... | 38   | NaN  | NaN  | NaN |
| 22 | CARL | Lake Carl Blackwell | OK | 36.15 | -97.29 | 2019 | 4  | 15 | 15 | 20 | ... | 36   | NaN  | NaN  | NaN |

```
idx = df['STID'].isin(['ACME', 'ALTU'])
df[idx]
```

|   | STID | NAME  | ST | LAT   | LON    | YR   | MO | DA | HR | MI | ... | RELH | CHIL | HEAT | WDD |
|---|------|-------|----|-------|--------|------|----|----|----|----|-----|------|------|------|-----|
| 0 | ACME | Acme  | OK | 34.81 | -98.02 | 2019 | 4  | 15 | 15 | 20 | ... | NaN  | NaN  | NaN  | NaN |
| 2 | ALTU | Altus | OK | 34.59 | -99.34 | 2019 | 4  | 15 | 15 | 20 | ... | 39   | NaN  | 82   | SSW |

# 28 Plotting

Plotting is an essential component of data analysis that enables researchers to effectively communicate data insights through visualizations. Python has several powerful libraries for creating a wide range of figures. Matplotlib, renowned for its vast gallery and versatility, is ideal for creating static and publication-quality figures. Bokeh is another library that excels in interactive and web-ready visualizations, making it perfect for dynamic data exploration. Seaborn is a library that was built on top of Matplotlib, and specializes in statistical graphics and provides a more high-level interface for creating sophisticated plots.

## 28.1 Dataset

To keep this plotting notebook simple, we will start by reading some daily environmental data recorded in a tallgrass prairie in the Kings Creek watershed, which is located within the Konza Prairie Biological Station near Manhattan, KS. The dataset includes the following variables:

| Variable Name | Units                          | Description                             | Sensor   |
|---------------|--------------------------------|---|----------|
| datetime      | -                              | Timestamp of the data record            |          |
| pressure      | kPa                            | Atmospheric pressure                    | Atmos 41 |
| tmin          | °C                             | Minimum temperature                     | Atmos 41 |
| tmax          | °C                             | Maximum temperature                     | Atmos 41 |
| tavg          | °C                             | Average temperature                     | Atmos 41 |
| rmin          | %                              | Minimum relative humidity               | Atmos 41 |
| rmax          | %                              | Maximum relative humidity               | Atmos 41 |
| prcp          | mm                             | Precipitation amount                    | Atmos 41 |
| srad          | MJ/m <sup>2</sup>              | Solar radiation                         | Atmos 41 |
| wspd          | m/s                            | Wind speed                              | Atmos 41 |
| wdir          | degrees                        | Wind direction                          | Atmos 41 |
| vpd           | kPa                            | Vapor pressure deficit                  | Atmos 41 |
| vwc_5cm       | m <sup>3</sup> /m <sup>3</sup> | Volumetric water content at 5 cm depth  | Teros 12 |
| vwc_20cm      | m <sup>3</sup> /m <sup>3</sup> | Volumetric water content at 20 cm depth | Teros 12 |
| vwc_40cm      | m <sup>3</sup> /m <sup>3</sup> | Volumetric water content at 40 cm depth | Teros 12 |
| soiltemp_5cm  | °C                             | Soil temperature at 5 cm depth          | Teros 12 |
| soiltemp_20cm | °C                             | Soil temperature at 20 cm depth         | Teros 12 |
| soiltemp_40cm | °C                             | Soil temperature at 40 cm depth         | Teros 12 |

| Variable Name | Units      | Description                       | Sensor |
|---------------|------------|-----------------------------------|--------|
| battv         | millivolts | Battery voltage of the datalogger |        |

```
# Import Numpy and Pandas modules
import numpy as np
import pandas as pd

# Read some tabulated weather data
df = pd.read_csv('../datasets/kings_creek_2022_2023_daily.csv', parse_dates=['datetime'])

# Display a few rows to inspect column headers and data
df.head(3)
```

|   | datetime   | pressure  | tmin  | tmax | tavg  | rmin      | rmax      | prcp | srad    | wspd     | wdir    |
|---|------------|-----------|-------|------|-------|-----------|-----------|------|---------|----------|---------|
| 0 | 2022-01-01 | 96.837917 | -14.8 | -4.4 | -9.6  | 78.475475 | 98.012496 | 0.0  | 2.09808 | 5.483333 | 0.96882 |
| 1 | 2022-01-02 | 97.994583 | -20.4 | -7.2 | -13.8 | 50.543218 | 84.935503 | 0.0  | 9.75636 | 2.216250 | 2.02327 |
| 2 | 2022-01-03 | 97.843750 | -9.4  | 8.8  | -0.3  | 40.622240 | 82.662479 | 0.0  | 9.68076 | 2.749167 | 5.66733 |

## 28.2 Matplotlib module

Matplotlib is a powerful and widely-used Python library for creating high-quality static and animated visualizations with a few lines of code that are suitable for scientific research. Matplotlib integrates well with other libraries like Numpy and Pandas, and can generate a wide range of graphs and has an extensive gallery of examples, so in this tutorial we will go over a few examples to learn the syntax, properties, and methods available to users in order to customize figures. To learn more visit [Matplotlib's official documentation](#).

### 28.2.1 Components of Matplotlib figures

- **Figure:** The entire window that everything is drawn on. The top-level container for all the elements.
- **Axes:** The part of the figure where the data is plotted, including any axes labeling, ticks, and tick labels. It's the area that contains the plot elements.
- **Plotting area:** The space where the data points are visualized. It's contained within the axes.
- **Axis:** These are the line-like objects and take care of setting the graph limits and generating the ticks and tick labels.

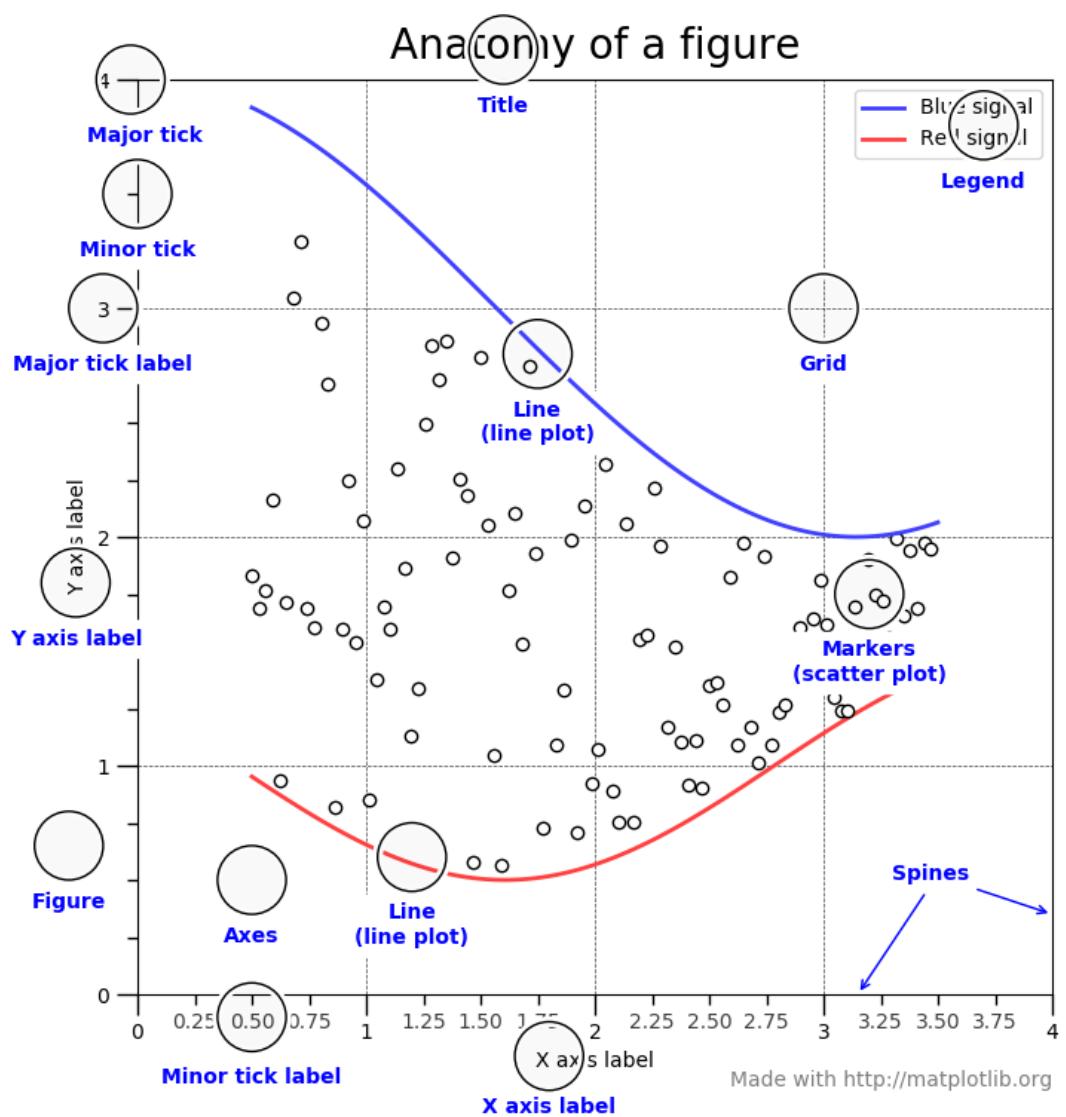


Figure 28.1: Components of a Matplotlib figure. Source:matplotlib.org

- **Ticks and Tick Labels:** The marks on the axis to denote data points and the labels assigned to these ticks.
- **Labels:** Descriptive text added to the  $x$  axis and  $y$  axis to identify what each represents.
- **Title:** A text label placed at the top of the axes to provide a summary or comment about the plot.
- **Legend:** A small area describing the different elements or data series of the plot. It's used to identify plots represented by different colors, markers, or line styles.

Essentially, a Matplotlib figure is an assembly of interconnected objects, each customizable through various properties and functions. When a figure is created, attributes such as the figure dimensions, axes properties, tick marks, font size of labels, and more come with pre-set default values. Understanding this object hierarchy is key to customize figures to your visualization needs.

### 28.2.2 Matplotlib syntax

Matplotlib has two syntax styles or interfaces for creating figures:

1. **function-based interface (easy)** that resembles Matlab's plotting syntax. This interface relies on using the `plt.____<function>_____` construction for adding/modifying each component of a figure. It is simpler and more straightforward than the object-based interface (see below), so the function-based style is sometimes easier for beginners or students with background in Matlab. The main disadvantage of this method is that it is implicit, meaning that the axes object (with all its attributes and methods) remains temporarily in the background since we are not saving it into a variable. This means that if we want to add/remove/modify something later on in one axes, we don't have that object available to us to implement the changes. One option is to get the current axes using `plt.gca()`, but we need to do this before adding another axes (say another subplot) to the figure. If you don't need to create sophisticated figures, then this method usually works just fine.
2. **object-based interface (advanced)** that offers more flexibility and control, particularly when dealing with multiple axes. In this interface, the `figure` and `axes` objects are explicit, meaning that each figure and axes object are stored as regular variables that provide the programmer access to all configuration options at any point in the script. The downside is that this syntax is a bit more verbose and sometimes less intuitive to beginners compared to the function-based approach. The official documentation typically favors the object-based syntax, so it is good to become familiar with it.

But don't panic, the syntax between these two methods is not that different. Below I added more syntax details, a cheat sheet to help you understand some of the differences, and several examples using real data. In [this article](#) you can learn more about the pros and cons of each style.

### 28.2.2.1 Function-based syntax

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]

# Create figure and plot
plt.figure(figsize=(4,4))
plt.plot(x, y)
plt.title("Simple Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

### 28.2.2.2 Object-based syntax

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]

# Create figure and axes
fig, ax = plt.subplots(figsize=(4,4))
ax.plot(x, y)
ax.set_title("Simple Line Plot")
ax.set_xlabel("X-axis")
ax.set_ylabel("Y-axis")
plt.show()
```

### 28.2.2.3 Matplotlib Cheat Sheet

| Operation     | function-based syntax | object-based syntax                                |
|---------------|-----------------------|--|
| Create figure | plt.figure()          | fig,ax = plt.subplots() fig,ax = plt.subplots(1,1) |

| Operation                      | function-based syntax   | object-based syntax  |
|--------------------------------|---|--|
| Simple line or scatter plot    | <code>plt.plot(x, y)</code><br><code>plt.scatter(x, y)</code>   | <code>ax.plot(x, y)</code><br><code>ax.scatter(x, y)</code>  |
| Add axis labels                | <code>plt.xlabel('label', fontsize=size)</code><br><code>plt.ylabel('label', fontsize=size)</code>        | <code>ax.set_xlabel('label', fontsize=size)</code><br><code>ax.set_ylabel('label', fontsize=size)</code>   |
| Change font size of tick marks | <code>plt.xticks(fontsize=size)</code><br><code>plt.yticks(fontsize=size)</code>                          | <del><code>plt.xticks(fontsize=size)</code></del><br><code>ax.set_xticks(fontsize=size)</code><br><code>ax.set_yticks(fontsize=size)</code>  |
| Add a legend                   | <code>plt.legend()</code>   | <code>ax.legend()</code>   |
| Remove tick marks              | <code>plt.tick_params(axis='both', which='both', bottom=False, top=False, labelbottom=False)</code>       | <code>ax.tick_params(axis='both', which='both', bottom=False, top=False, labelbottom=False)</code>   |
| Add a title                    | <code>plt.title('title')</code>   | <code>ax.set_title('title')</code>   |
| Add a secondary axis           | <code>plt.twinx()</code>  | <code>ax_secondary = ax.twinx()</code>   |
| Rotate tick labels             | <code>plt.xticks(rotation=angle)</code><br><code>plt.yticks(rotation=angle)</code>                        | <del><code>plt.xticks(rotation=angle)</code></del><br><code>ax.tick_params(axis='y', rotation=angle)</code>  |
| Change scale                   | <code>plt.xscale('log')</code><br><code>plt.yscale('log')</code>  | <code>ax.set_xscale('log')</code><br><code>ax.set_yscale('log')</code>   |
| Change axis limits             | <code>plt.xlim([xmin, xmax])</code><br><code>plt.ylim([ymin, ymax])</code>                                | <code>ax.set_xlim([xmin, xmax])</code><br><code>ax.set_ylim([ymin, ymax])</code>   |
| Create subplots                | <code>plt.subplots(1, 2, 1)</code><br><code>plt.subplots(2, 2, 1)</code>                                  | <del><code>fig, (ax1, ax2) = plt.subplots(1, 2)</code></del><br><del><code>fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)</code></del><br><code>fig, axs = plt.subplots(2, 2); axs[0, 0].plot(x, y)</code> |
| Change xaxis dateformat        | <code>fmt = mdates.DateFormatter('%b-%y')</code><br><code>plt.gca().xaxis.set_major_formatter(fmt)</code> | <code>fmt = mdates.DateFormatter('%b-%y')</code><br><code>ax.xaxis.set_major_formatter(fmt)</code>   |

```
# Import matplotlib modules
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
```

### 28.2.3 Line plot

A common plot when working with meteorological data is to show maximum and minimum air temperature.

```
# Create figure
plt.figure(figsize=(8,4)) # If you set dpi=300 the figure is much better quality

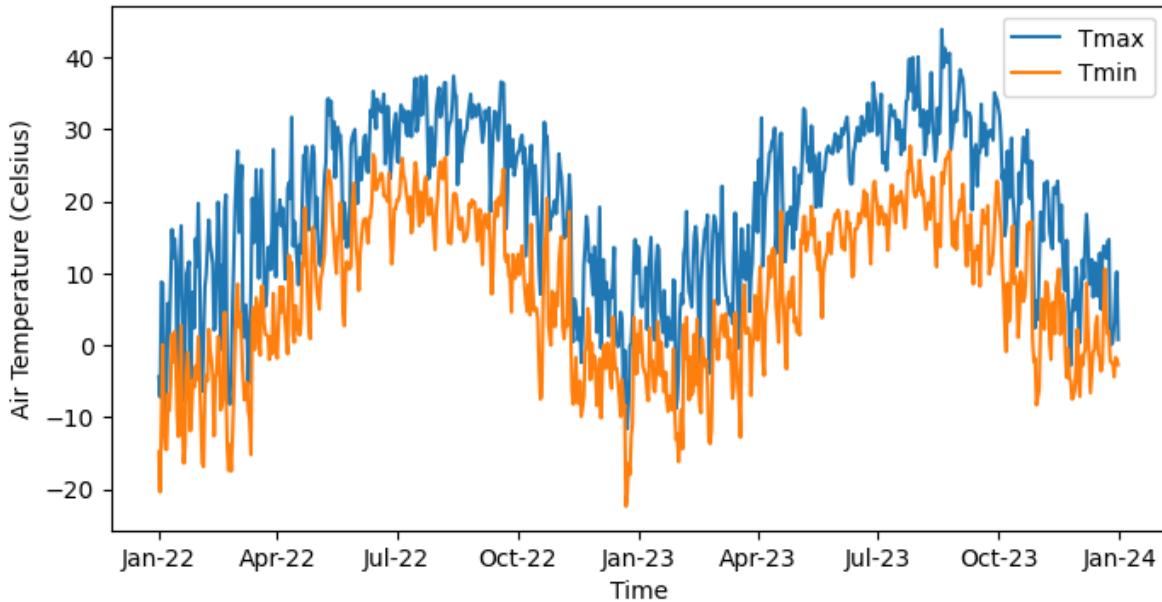
# Add lines to axes
plt.plot(df['datetime'], df['tmax'], label='Tmax')
plt.plot(df['datetime'], df['tmin'], label='Tmin')

# Customize chart attributes
plt.xlabel('Time', fontsize=10)
plt.ylabel('Air Temperature (Celsius)', fontsize=10)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.legend(fontsize=10)

# Create custom dateformat for x-axis
date_format = mdates.DateFormatter('%b-%y')

# We don't have the axes object saved into a variable, so to set the date format
# we need to get the current axes (gca).
# If we were adding more axes to this figure, then gca() will return the latest axes
plt.gca().xaxis.set_major_formatter(date_format)

# Save figure (check for more options in the Matplotlib's documentation)
plt.savefig('line_plot.jpg', dpi=300, facecolor='w', pad_inches=0.1) # Call before plt.show()
plt.show()
```



#### 28.2.3.1 Object-based code

```

fig, ax = plt.subplots(1,1,figsize=(8,4), dpi=300)
ax.plot(df['datetime'], df['tmax'], label='Tmax')
ax.plot(df['datetime'], df['tmin'], label='Tmin')
ax.set_xlabel('Time', fontsize=10)
ax.set_ylabel('Air Temperature (Celsius)', fontsize=10)
ax.tick_params(axis='both', labelsize=10)
ax.legend(fontsize=10)
date_format = mdates.DateFormatter('%b-%y')

# Here we have the axes object saved into the `ax` variable, so it is explicit and
# we just need to access the method within the object.
# We could do this step later, even if we create other axes with different variable names.
ax.xaxis.set_major_formatter(date_format)

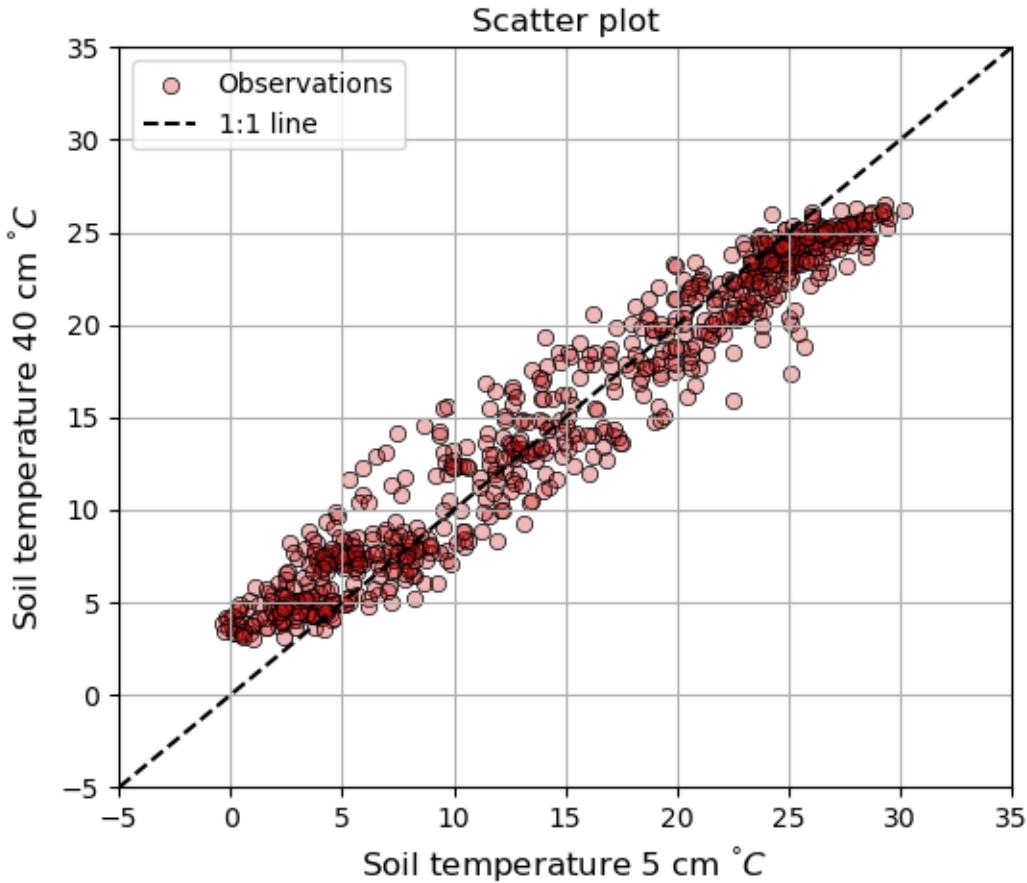
# Save figure
plt.savefig('line_plot.jpg', dpi=300, facecolor='w', pad_inches=0.1) # Call before plt.show()
plt.show()

```

#### 28.2.4 Scatter plot

Let's inspect soil temperature data at 5 and 40 cm depth and see how similar or different these two variables are. A 1:1 line will serve as the reference of perfect equality.

```
plt.figure(figsize=(6,5))
plt.scatter(df['soiltemp_5cm'], df['soiltemp_40cm'],
            marker='o', facecolor=(0.8, 0.1, 0.1, 0.3),
            edgecolor='k', linewidth=0.5, label='Observations')
plt.plot([-5, 35], [-5, 35], linestyle='--', color='k', label='1:1 line') # 1:1 line
plt.title('Scatter plot', fontsize=12, fontweight='normal')
plt.xlabel('Soil temperature 5 cm $^\circ\text{C}$', size=12)
plt.ylabel('Soil temperature 40 cm $^\circ\text{C}$', size=12)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.xlim([-5, 35])
plt.ylim([-5, 35])
plt.legend(fontsize=10)
plt.grid()
plt.show()
```



#### 28.2.4.1 Object-based syntax

```

fig, ax = plt.subplots(1, 1, figsize=(6,5), edgecolor='k')
ax.scatter(df['soiltemp_5cm'], df['soiltemp_40cm'],
           marker='o', facecolor=(0.8, 0.1, 0.1, 0.3),
           edgecolor='k', linewidth=0.5, label='Observations')
ax.plot([-5, 35], [-5, 35], linestyle='--', color='k', label='1:1 line')
ax.set_title('Scatter plot', fontsize=12, fontweight='normal')
ax.set_xlabel("Soil temperature 5 cm $^\circ\text{C}$", size=12)
ax.set_ylabel("Soil temperature 40 cm $^\circ\text{C}$", size=12)
ax.set_xlim([-5, 35])
ax.set_ylim([-5, 35])
ax.tick_params(axis='both', labelsize=12)
ax.grid(True)

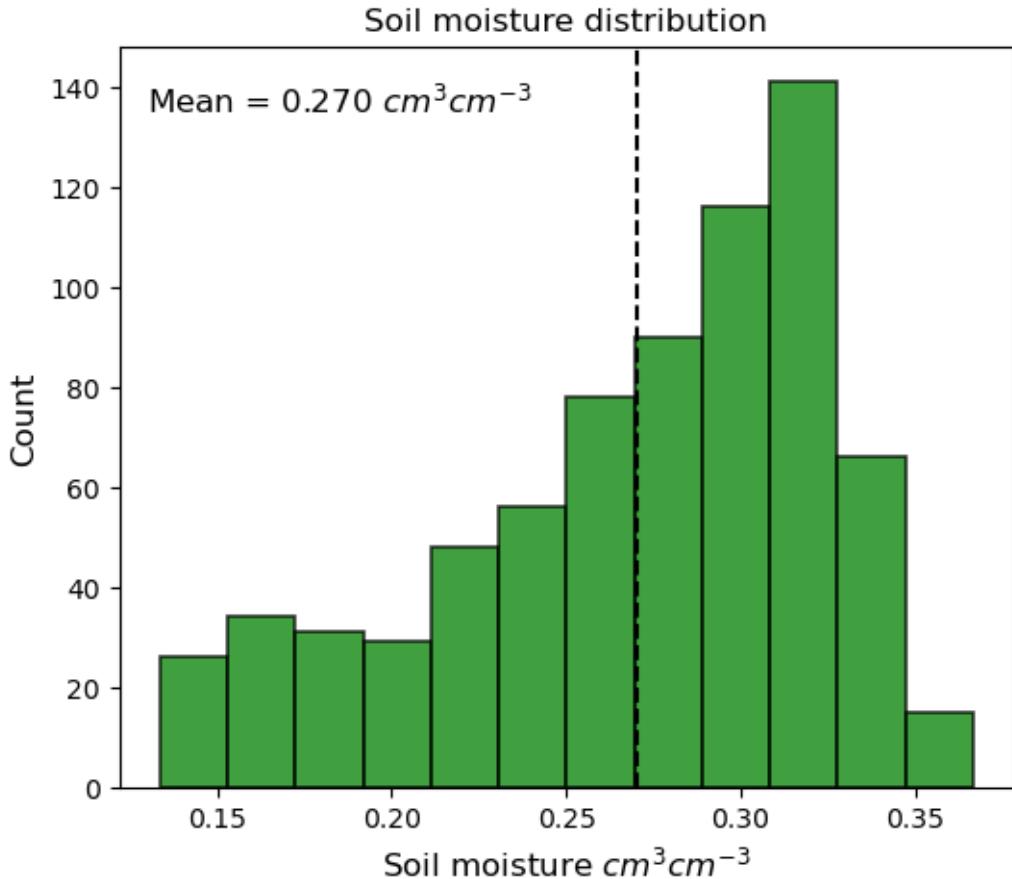
```

```
plt.show()
```

### 28.2.5 Histogram

One of the most common and useful charts to describe the distribution of a dataset is the histogram.

```
plt.figure(figsize=(6,5))
plt.hist(df['vwc_5cm'], bins='scott', density=False,
         facecolor='g', alpha=0.75, edgecolor='black', linewidth=1.2)
plt.title('Soil moisture distribution', fontsize=12)
plt.xlabel('Soil moisture $cm^3 cm^{-3}$', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.text(0.13, 135, "Mean = " + f"{df['vwc_5cm'].mean():.3f}" + " $cm^3 cm^{-3}$", size=12)
plt.axvline(df['vwc_5cm'].mean(), linestyle='--', color='k')
plt.show()
```



#### 28.2.5.1 Object-based syntax

```

fig, ax = plt.subplots(figsize=(6,5))
ax.hist(df['vwc_5cm'], bins='scott', density=False,
        facecolor='g', alpha=0.75, edgecolor='black', linewidth=1.2)

ax.text(0.13, 135, "Mean = " + f"{df['vwc_5cm'].mean():.3f}" + " $cm^3\ cm^{-3}$", size=12)
ax.set_xlabel('Volumetric water content $cm^3\ cm^{-3}$', fontsize=12)
ax.set_ylabel('Count', fontsize=12)
ax.tick_params('both', labelsize=12)
ax.set_title('Soil moisture distribution', fontsize=12)
ax.axvline(df['vwc_5cm'].mean(), linestyle='--', color='k')
plt.show()

```

## 28.2.6 Subplots

In fields like agronomy, environmental science, hydrology, and meteorology sometimes we want to show multiple variables in one figure, but in different charts. Other times we want to show the same variable, but in separate charts for different locations or sites. In Matplotlib we can achieve this using subplots.

```
# Define date format
date_format = mdates.ConciseDateFormatter(ax.xaxis.get_major_locator())

# Create figure
plt.figure(figsize=(10,6))

# Set width and height spacing between subplots
plt.subplots_adjust(wspace=0.3, hspace=0.4)

# Add superior title for entire figure
plt.suptitle('Kings Creek temperatures 2022-2023')

# Subplot 1 of 4
plt.subplot(2, 2, 1)
plt.plot(df["datetime"], df["tavg"])
plt.title('Air temperature', fontsize=12)
plt.ylabel('Temperature', fontsize=12)
plt.ylim([-20, 40])
plt.gca().xaxis.set_major_formatter(date_format)
plt.text(df['datetime'].iloc[0], 32, 'A', fontsize=14)

# Subplot 2 of 4
plt.subplot(2, 2, 2)
plt.plot(df["datetime"], df["soiltemp_5cm"])
plt.title('Soil temperature 5 cm', size=12)
plt.ylabel('Temperature', size=12)
plt.ylim([-20, 40])
plt.gca().xaxis.set_major_formatter(date_format)
plt.text(df['datetime'].iloc[0], 32, 'B', fontsize=14)

# Subplot 3 of 4
plt.subplot(2, 2, 3)
plt.plot(df["datetime"], df["soiltemp_20cm"])
plt.title('Soil temperature 20 cm', size=12)
plt.ylabel('Temperature', size=12)
```

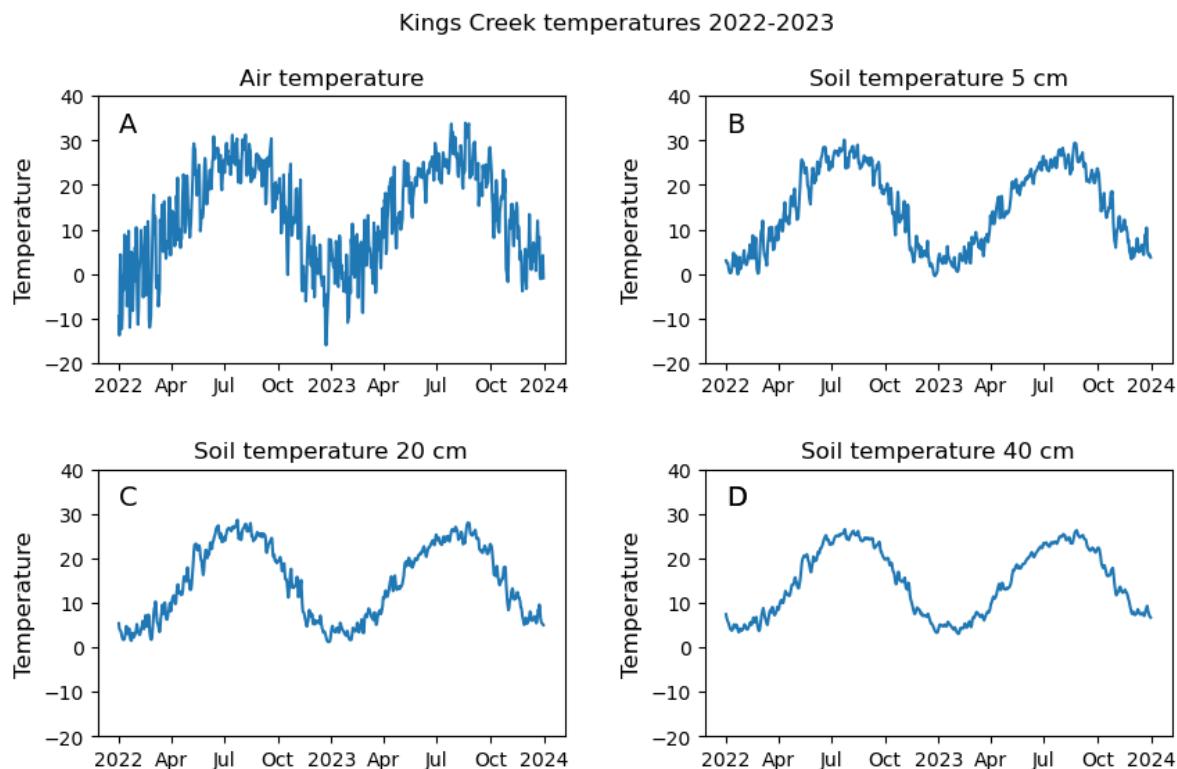
```

plt.ylim([-20, 40])
plt.gca().xaxis.set_major_formatter(date_format)
plt.text( df['datetime'].iloc[0], 32, 'C', fontsize=14)

# Subplot 4 of 4
plt.subplot(2, 2, 4)
plt.text( df['datetime'].iloc[0], 32, 'D', fontsize=14)
plt.plot(df["datetime"], df["soiltemp_40cm"])
plt.title('Soil temperature 40 cm', size=12)
plt.ylabel('Temperature', size=12)
plt.ylim([-20, 40])
plt.gca().xaxis.set_major_formatter(date_format)
plt.text( df['datetime'].iloc[0], 32, 'D', fontsize=14)

plt.show()

```



### 28.2.6.1 Object-based syntax

```
# Define date format
date_format = mdates.ConciseDateFormatter(ax.xaxis.get_major_locator())

# Create figure
fig, ((ax1,ax2),(ax3,ax4)) = plt.subplots(2, 2, figsize=(10,6))

# Set width and height spacing between subplots
fig.subplots_adjust(wspace=0.3, hspace=0.4)

# Add superior title for entire figure
fig.suptitle('Kings Creek temperatures 2022-2023')

ax1.set_title('Air temperature', size=12)
ax1.set_ylabel('Temperature', size=12)
ax1.set_ylim([-20, 40])
ax1.xaxis.set_major_formatter(date_format)
ax1.text( df['datetime'].iloc[0], 32, 'A', fontsize=14)

ax2.set_title('Soil temperature 5 cm', size=12)
ax2.set_ylabel('Temperature', size=12)
ax2.set_ylim([-20, 40])
ax2.xaxis.set_major_formatter(date_format)
ax2.text( df['datetime'].iloc[0], 32, 'B', fontsize=14)

ax3.set_title('Soil temperature 20 cm', size=12)
ax3.set_ylabel('Temperature', size=12)
ax3.set_ylim([-20, 40])
ax3.xaxis.set_major_formatter(date_format)
ax3.text( df['datetime'].iloc[0], 32, 'C', fontsize=14)

ax4.set_title('Soil temperature 40 cm', size=12)
ax4.set_ylabel('Temperature', size=12)
ax4.set_ylim([-20, 40])
ax4.xaxis.set_major_formatter(date_format)
ax4.text( df['datetime'].iloc[0], 32, 'D', fontsize=14)

# ----- READ THIS -----
# To illustrate the power of the object-based notation I set
# all the line plots here at the end. In the function-based syntax you are forced
```

```

# to set all the elements and attributes within the block of code for that subplot
ax1.plot(df["datetime"], df["tavg"])
ax2.plot(df["datetime"], df["soiltemp_5cm"])
ax3.plot(df["datetime"], df["soiltemp_20cm"])
ax4.plot(df["datetime"], df["soiltemp_40cm"])

plt.show()

```

### 28.2.7 Secondary Y axis plots

Sometimes we want to show two related variables with different range or entirely different units in the same chart. In this case, two chart axes can share the same x-axis but have two different y-axes. A typical example of this consists of displaying soil moisture variations together with precipitation. While less common, it is also possible for two charts to share the same y-axis and have two different x-axes.

```

# Creating plot with secondary y-axis
plt.figure(figsize=(8,4))

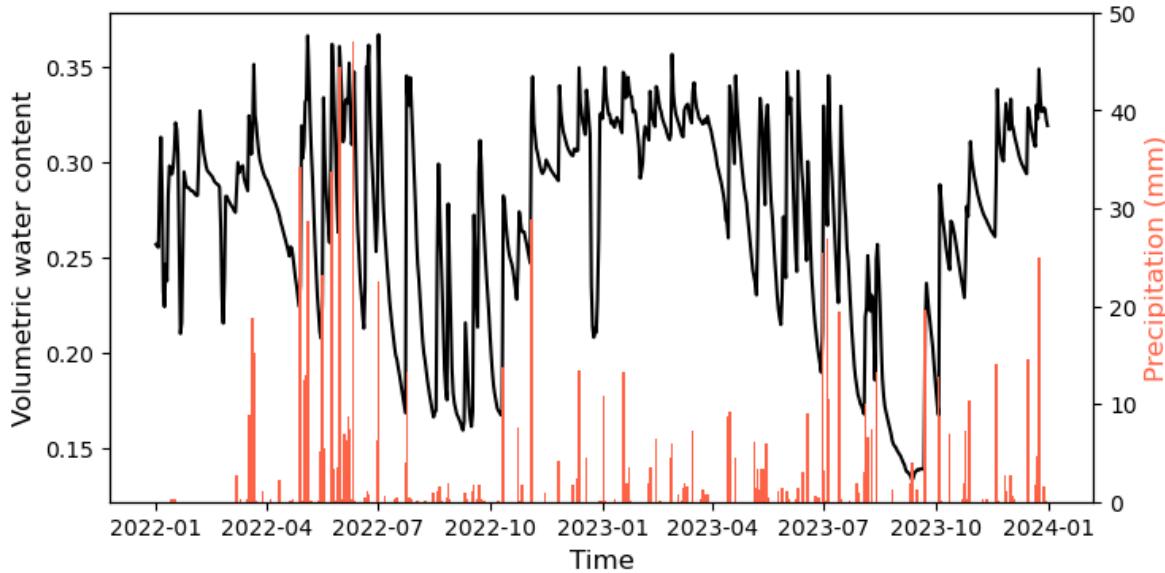
plt.plot(df["datetime"], df["vwc_5cm"], '-k')
plt.xlabel('Time', size=12)
plt.ylabel('Volumetric water content', color='k', size=12)

plt.twinx()

plt.bar(df["datetime"], df["prcp"], width=2, color='tomato', linestyle='--')
plt.ylabel('Precipitation (mm)', color='tomato', size=12)
plt.ylim([0, 50])

plt.show()

```



#### 28.2.7.1 Object-based syntax

```
# Creating plot with secondary y-axis
fig, ax = plt.subplots(figsize=(8,4), facecolor='w')

ax.plot(df["datetime"], df["vwc_5cm"], '-k')
ax.set_xlabel('Time', size=12)
ax.set_ylabel('Volumetric water content', color='k', size=12)

ax2 = ax.twinx()

ax2.bar(df["datetime"], df["prcp"], width=2, color='tomato', linestyle='--')
ax2.set_ylabel('Precipitation (mm)', color='tomato', size=12)
ax2.set_ylim([0, 50])

plt.show()

#### Themes
```

In addition to the default style, Matplotlib also offers a variety of pre-defined themes. To see some examples visit the following websites:

Gallery 1 at: [https://matplotlib.org/gallery/style\\_sheets/style\\_sheets\\_reference.html](https://matplotlib.org/gallery/style_sheets/style_sheets_reference.html)

Gallery 2 at: [https://tonysyu.github.io/raw\\_content/matplotlib-style-gallery/gallery.html](https://tonysyu.github.io/raw_content/matplotlib-style-gallery/gallery.html)

```

# Run this line to see all the styling themes available
print(plt.style.available)

['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-nogrid', 'bmh', 'cl

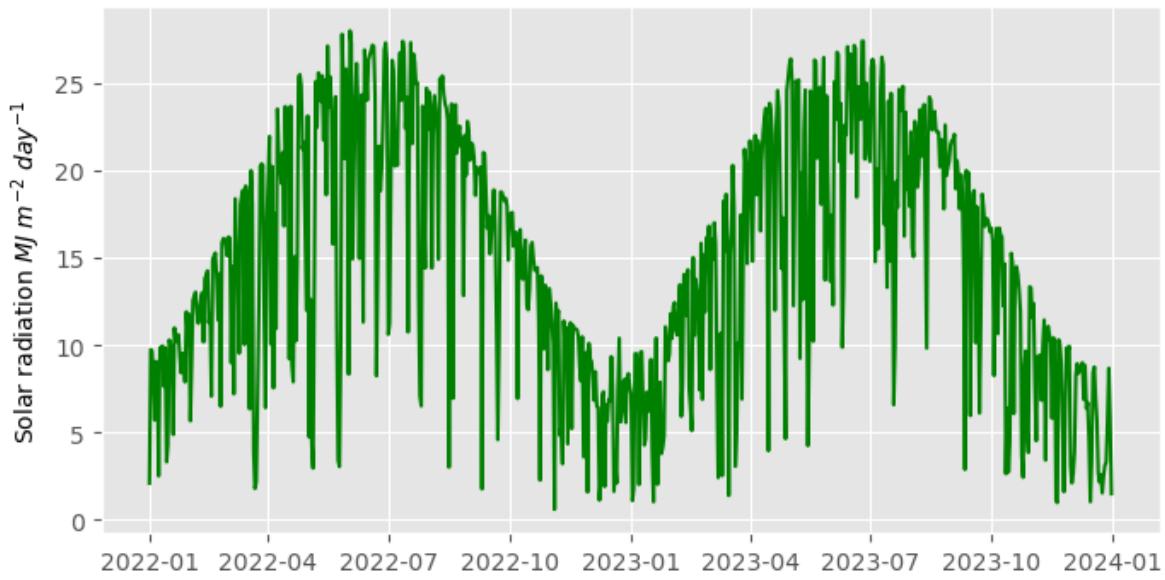
# Change plot defaults to ggplot style (similar to ggplot R language library)
# Use plt.style.use('default') to revert style

plt.style.use('ggplot') # Use plt.style.use('default') to revert.

plt.figure(figsize=(8,4))
plt.plot(df["datetime"], df["srad"], '-g')
plt.ylabel("Solar radiation $MJ \ m^{-2} \ day^{-1}$")

plt.show()

```



## 28.3 Bokeh module

The [Bokeh](#) plotting library was designed for creating interactive visualizations for modern web browsers. Compared to Matplotlib, which excels in creating static plots, Bokeh emphasizes interactivity, offering tools to create dynamic and interactive graphics. Additionally, Bokeh

integrates well with the Pandas library and provides a consistent and standardized syntax. This focus on interactivity and ease of use makes Bokeh well suited for web-based data visualizations and applications.

Unlike Matplotlib, where you typically import the entire library with a single command, Bokeh is organized into various sub-modules catered to different functionalities. This structure means that you import specific components from their respective modules, which aligns with the functionality you intend to use. While this might require a bit more upfront learning about the library, it also means that you are only importing what you need.

```
# Import Bokeh modules
from bokeh.plotting import figure, output_notebook, show
from bokeh.models import HoverTool

# Initialize the bokeh for the notebook
# If everything went correct you should see the Bokeh icon displaying below.
output_notebook()
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_l

## 28.4 Interactive line plot

```
# Create figure
p = figure(plot_width=800, plot_height=400, title='Kings Creek', x_axis_type='datetime')

# Add line to the figure
p.line(df['datetime'], df['tmin'], line_color='blue', line_width=2, legend_label='Tmin')

# Add another line. In this case I used a different, but equivalent, syntax
# This syntax leverages the dataframe and its column names
p.line(source=df, x='datetime', y='tmax', line_color='tomato', line_width=2, legend_label='Tmax')

# Customize figure properties
p.xaxis.axis_label = 'Time'
p.yaxis.axis_label = 'Air Temperature (Celsius)'

# Set the font size of the x-axis and y-axis labels
p.yaxis.axis_label_text_font_size = '12pt' # Defined as a string using points simialr to W
```

```

p.xaxis.axis_label_text_size = '12pt'

# Set up the size of the labels in the major ticks
p.xaxis.major_label_text_size = '12pt'
p.yaxis.major_label_text_size = '12pt'

# Add legend
p.legend.location = "top_left"
p.legend.title = "Legend"
p.legend.label_text_font_style = "italic"
p.legend.label_text_color = "black"
p.legend.border_line_width = 1
p.legend.border_line_color = "navy"
p.legend.border_line_alpha = 0.8
p.legend.background_fill_color = "white"
p.legend.background_fill_alpha = 0.9

# Hover tool for interactive tooltips on mouse hover over the plot
p.add_tools(HoverTool(tooltips=[("Date:", "$x{%F}"), ("Temperature:", "$y{%.1f} Celsius")],
                      formatters={'$x':'datetime', '$y':'printf'},
                      mode='mouse'))

# Display figure
show(p)

```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_e

## 28.5 Seaborn module

Seaborn is a plotting library based on Matplotlib, specifically tailored for statistical data visualization. It stands out in scientific research for its ability to create informative and attractive statistical graphics with ease. Seaborn integrates well with Pandas DataFrames and its default styles and color palettes are designed to be aesthetically pleasing and ready for publication. Seaborn offers complex visualizations like heatmaps, violin plots, boxplots, and matrix scatter plots.

```

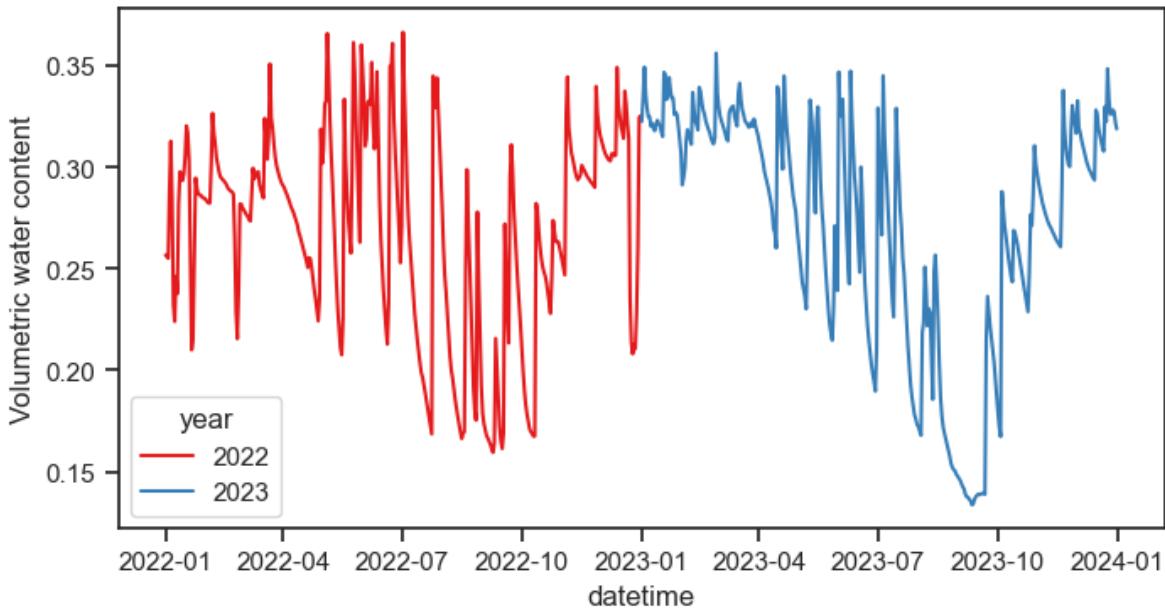
# Import Seaborn module
import seaborn as sns
sns.set_theme(style="ticks")

```

### 28.5.1 Line plot

```
# Basic line plot
df_subset = df[['datetime', 'vwc_5cm', 'vwc_20cm', 'vwc_40cm']].copy()
df_subset['year'] = df_subset['datetime'].dt.year

plt.figure(figsize=(8,4))
sns.lineplot(data=df_subset, x='datetime', y='vwc_5cm', hue='year', palette='Set1')
plt.ylabel('Volumetric water content')
plt.show()
```



### 28.5.2 Correlation matrix

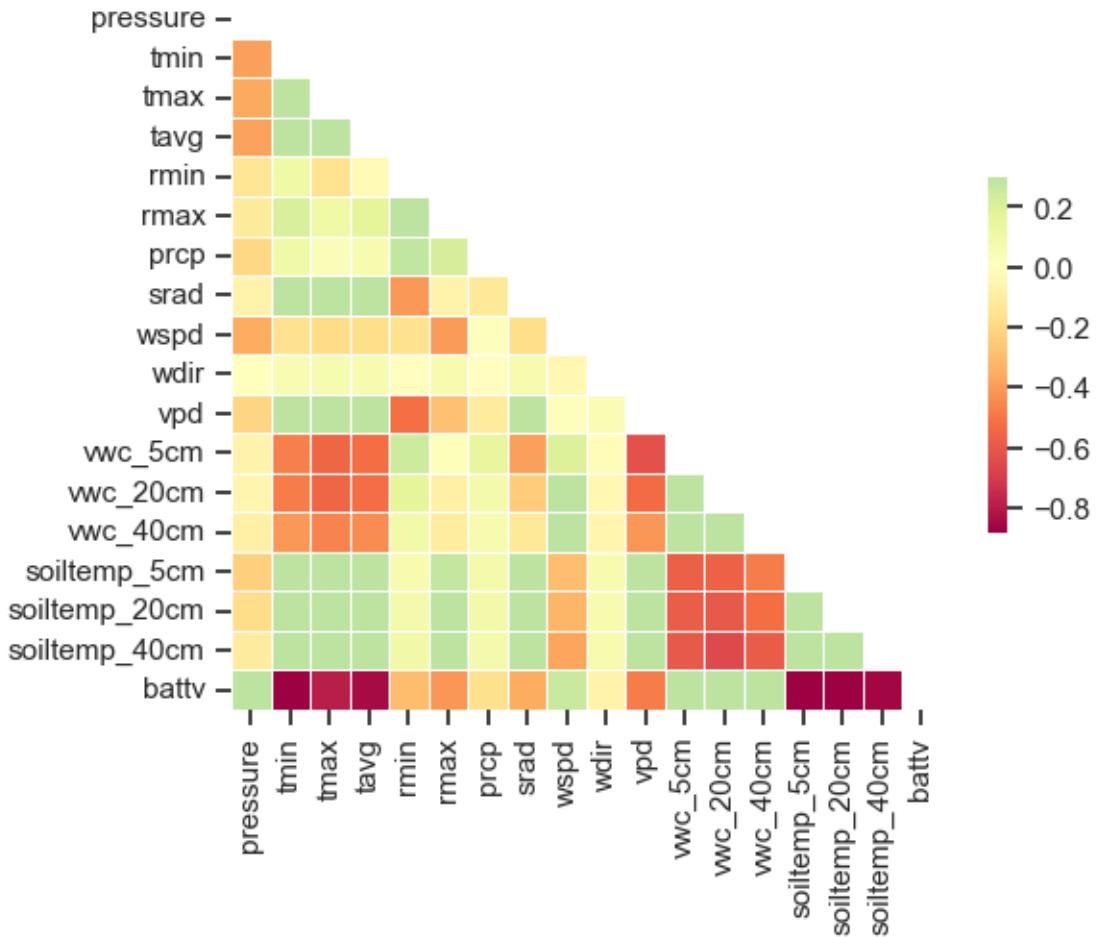
```
# Compute the correlation matrix
corr = df.corr(numeric_only=True)

# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap='Spectral', vmax=.3, center=0,
```

```
square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

<Axes: >



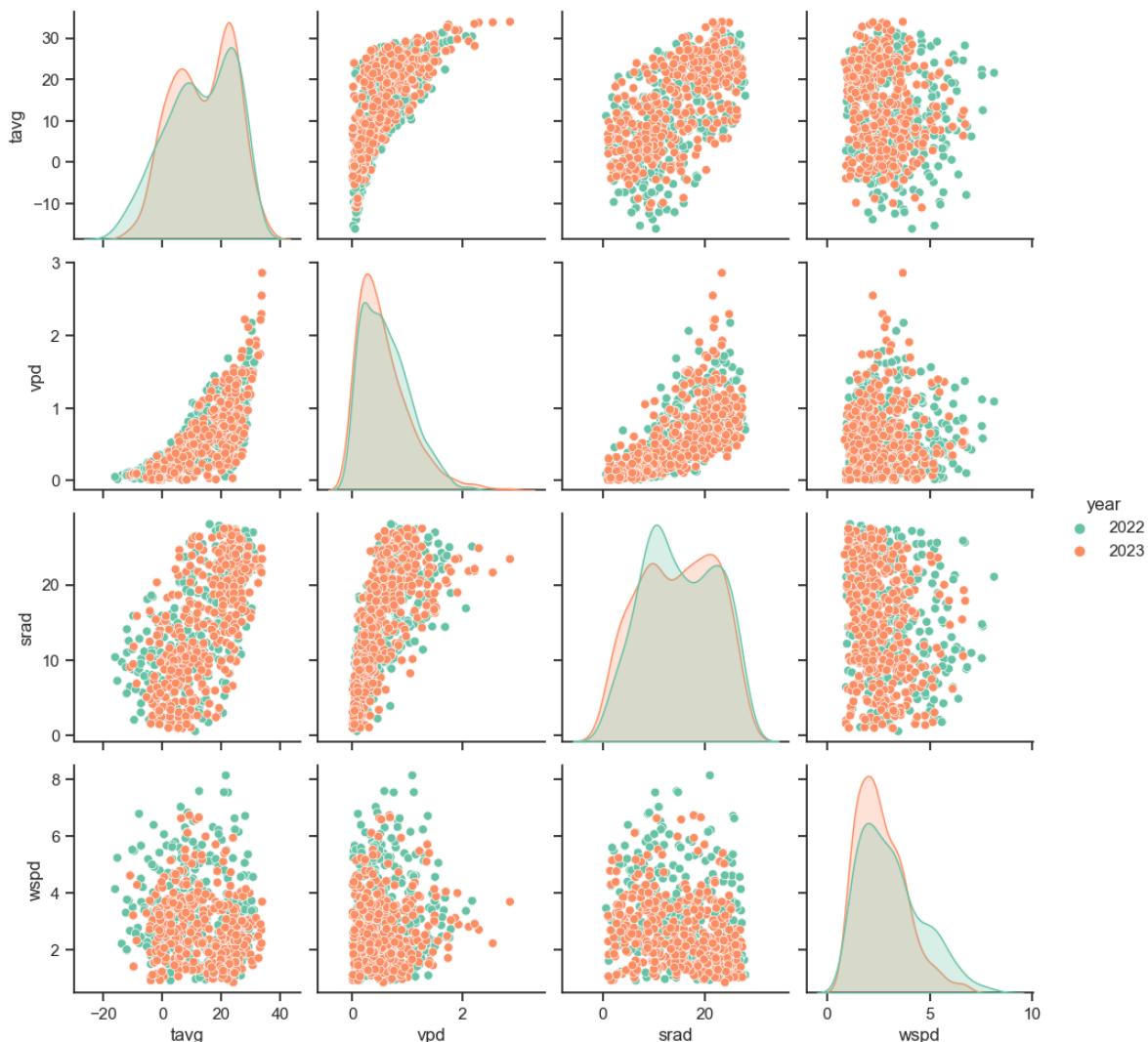
### 28.5.3 Scatterplot matrix

```
# Create subset of main dataframe
df_subset = df[['datetime', 'tavg', 'vpd', 'srad', 'wspd']].copy()
df_subset['year'] = df_subset['datetime'].dt.year

plt.figure(figsize=(8,6))
sns.pairplot(df_subset, hue="year", palette="Set2")
```

```
plt.show()
```

<Figure size 800x600 with 0 Axes>



#### 28.5.4 Heatmap

Visualization of air and soil temperature at 5, 20, and 40 cm depths on a weekly basis.

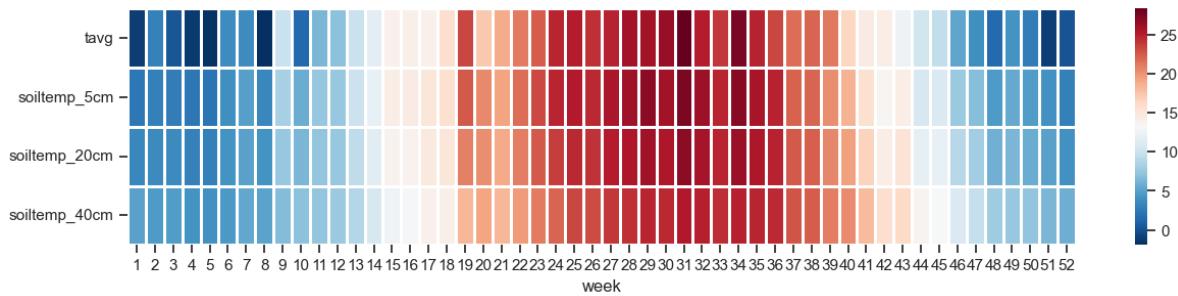
```

# Summarize data by month
df_subset = df[['datetime', 'tavg', 'soiltemp_5cm', 'soiltemp_20cm', 'soiltemp_40cm']].copy()
df_subset['week'] = df['datetime'].dt.isocalendar().week

# Average the values for both years on a weekly basis
df_subset = df_subset.groupby(["week"]).mean(numeric_only=True).round(2)

# Create Heatmap
plt.figure(figsize=(15,3))
sns.heatmap(df_subset.T, annot=False, linewidths=1, cmap="RdBu_r")
plt.show()

```



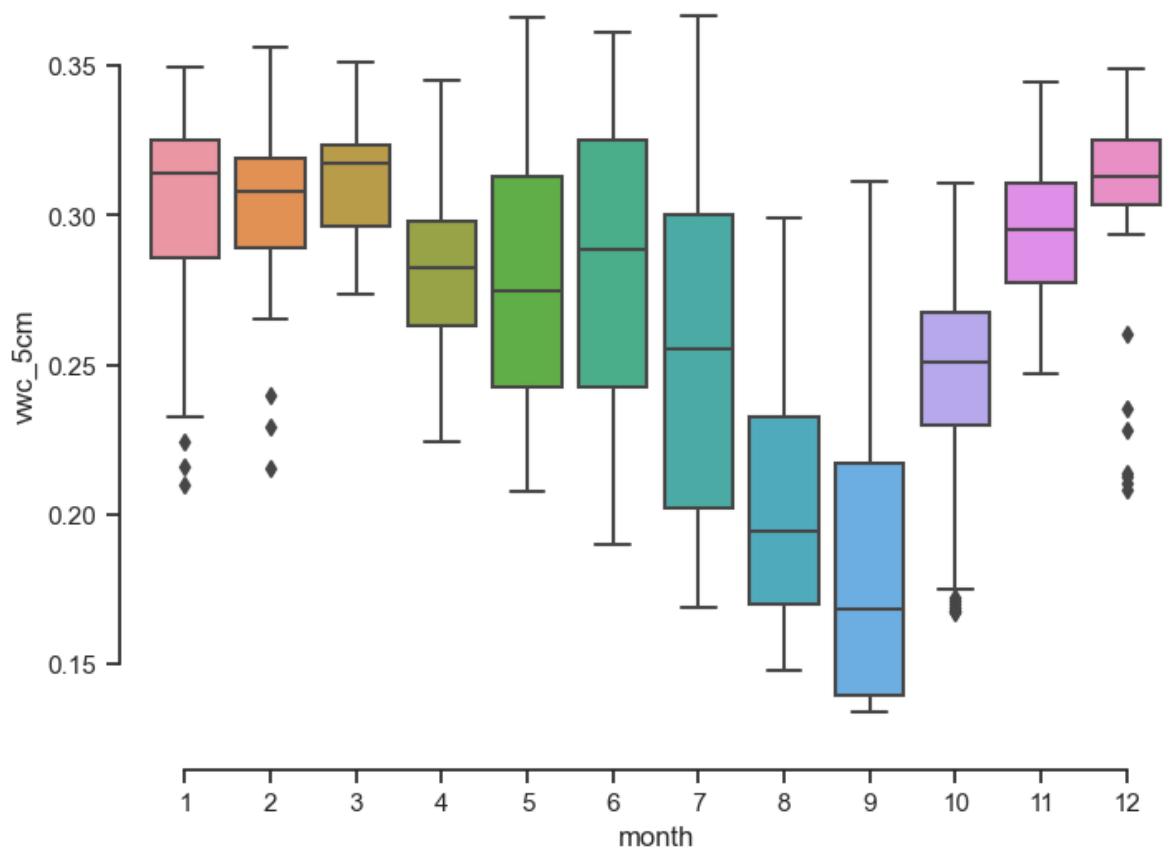
### 28.5.5 Boxplot

```

# Summarize data by month
df_subset = df[['datetime', 'vwc_5cm']].copy()
df_subset['month'] = df['datetime'].dt.month

# Draw a nested boxplot to show bills by day and time
plt.figure(figsize=(8,6))
sns.boxplot(data=df_subset, x="month", y="vwc_5cm")
sns.despine(offset=10, trim=True)
plt.show()

```



# 29 Widgets

Adding interactivity to Jupyter notebooks can be a great way of improving exploratory data analysis and real-time interactive data visualization. The `ipywidgets` library provides a powerful toolset for creating interactive widget elements like sliders, dropdowns, and buttons in Jupyter notebooks.

The following examples will demonstrate:

- the basic syntax to create widgets and set custom parameters
- how to define a function that ingests widget values
- how to connect the widget and the function

## Note

The widgets may not appear in the online version of the tutorial. Either copy the code and run it in your local Jupyter notebook or download the notebook from the Github repository.

## 29.1 Example 1: Convert bushels to metric tons

A common task in agronomy is to convert grain yields from bushels to metric tons. We will use `ipywidgets` to create interactive dropdowns and slider widgets to facilitate the conversion between these two units for common crops.

```
# Import modules
import ipywidgets as widgets

# Define widget
crop_dropdown = widgets.Dropdown(options=['Barley', 'Corn', 'Sorghum', 'Soybeans', 'Wheat'],
                                  value='Wheat', description='Crop')
bushels_slider = widgets.FloatSlider(value=40, min=0, max=200, step=1,
                                      description='Bushels/acre')

# Define function
```

```

def bushels_to_tons(crop, bu):
    """Function that converts bushels to metric tons for common crops.
       Source: https://grains.org/
    """
    # Define constants
    lbs_per_ton = 0.453592/1000
    acres_per_ha = 2.47105

    # Convert bu -> lbs -> tons
    if crop == 'Barley':
        tons = bu * 48 * lbs_per_ton

    elif crop == 'Corn' or crop == 'Sorghum':
        tons = bu * 56 * lbs_per_ton

    elif crop == 'Wheat' or crop == 'Soybeans':
        tons = bu * 60 * lbs_per_ton

    # Convert acre -> hectares
    tons = round(tons * acres_per_ha, 2)
    return widgets.FloatText(value=tons, description='Tons/ha', disabled=True)

# Define interactivity
widgets.interact(bushels_to_tons, crop=crop_dropdown, bu=bushels_slider);

interactive(children=(Dropdown(description='Crop', index=4, options=('Barley', 'Corn', 'Sorg

```

## 29.2 Example 2: Runoff-Precipitation

Widgets are also a great tool to explore and learn how models representing real-world processes respond to input changes. In this example we will explore how the curve number of a soil is related to the amount of runoff for a range of precipitation amounts.

```

import ipywidgets as widgets
import matplotlib.pyplot as plt
import numpy as np
plt.style.use('ggplot')

```

```

# Create widget
cn_slider = widgets.IntSlider(50, min=1, max=100, description='Curve number')

# Create function
def estimate_runoff(cn):
    """Function that computes runoff based on the
    curve number method proposed by the Soil Conservation Service
    Source: https://www.wikiwand.com/en/Runoff_curve_number

    Inputs
    cn : Curve number. 0 means fully permeable and 100 means fully impervious.

    Returns
    Figure of runoff as a function of precipitation
    """
    P = np.arange(0, 12, step=0.01) # Precipitation in inches
    R0 = np.zeros_like(P)
    S = 1000/cn - 10
    Ia = S * 0.05 # Initial abstraction (inches)
    idx = P > Ia
    R0[idx] = (P[idx] - Ia)**2 / (P[idx] - Ia + S)

    # Create figure
    plt.figure(figsize=(6,4))
    plt.plot(P,R0,'-k')
    plt.title('Curve Number Method')
    plt.xlabel('Precipitation (inches)')
    plt.ylabel('Runoff (inches)')
    plt.xlim([0,12])
    plt.ylim([0,12])
    return plt.show()

# Define interactivity
widgets.interact(estimate_runoff, cn=cn_slider);

interactive(children=(IntSlider(value=50, description='Curve number', min=1), Output()), _dominant=True)

```

# 30 SQLite Database

SQLite is a lightweight, disk-based database that doesn't require a separate server. Python comes with a built-in `sqlite3` module, which allows you to work with SQLite databases. SQL stands for Structured Query Language, which is a standardized programming language that is used to manage relational databases. To learn more about SQLite databases, in this tutorial we will:

1. create a simple database by adding and removing a few entries. This example illustrates how to start a new database from scratch without prior tabulated data.
2. create a more advanced database using a real-world dataset and the Pandas library. This example illustrates how to convert an existing spreadsheet with data into a database that you can then continue to add, update, modify, or delete data entries.

## 30.1 Additional software

To access and inspect the database I recommend using an open source tool like [sqlitebrowser](#). With this tool you can also create, design, and edit sqlite databases, but we will do some of these steps using Python.

## 30.2 Key commands

- **CREATE TABLE:** A SQL command used to create a new table in a database.
- **INSERT:** A SQL command used to add new rows of data to a table in the database.
- **SELECT:** A SQL command used to query data from a table, returning rows that match the specified criteria.
- **UPDATE:** A SQL command used to modify existing data in a table.
- **DELETE:** A SQL command used to remove rows from a table in the database.

## 30.3 Data types

SQLite supports a variety of data types:

- **TEXT**: For storing character data. SQLite supports UTF-8, UTF-16BE, and UTF-16LE encodings.
- **INTEGER**: For storing integer values. The size can be from 1 byte to 8 bytes, depending on the magnitude of the value.
- **REAL**: For storing floating-point values. It is a double-precision (8-byte) floating point number.
- **BLOB**: Stands for Binary Large Object. Used to store data exactly as it was input, such as images, files, or binary data.
- **NUMERIC**: This type can be used for both integers and floating-point numbers. SQLite decides whether to use integer or real based on the value's nature.
- **BOOLEAN**: SQLite does not have a separate boolean storage class. Instead, boolean values are stored as integers 0 (false) and 1 (true).
- **DATE and TIME**: SQLite does not have a storage class set aside for storing dates and/or times. Instead, they are stored as TEXT (as ISO8601 strings), REAL (as Julian day numbers), or INTEGER (as Unix Time, the number of seconds since 1970-01-01 00:00:00 UTC).

In SQLite the datatype you specify for a column acts more like a hint than a strict enforcement, allowing for flexibility in the types of data that can be inserted into a column. This is a distinctive feature compared to more rigid type systems in other database management systems.

## 30.4 Set up a simple database

After importing the `sqlite3` module we create a connection to a new SQLite database. If the file doesn't exist, the SQLite module will create it. This is convenient since we don't have to be constantly checking whether the database exists or worry about overwriting the database.

```
# Import modules
import sqlite3

# Connect to the database
conn = sqlite3.connect('soils.db')
```

```

# Create a cursor object using the cursor() method
cursor = conn.cursor()

# Create table
cursor.execute('''CREATE TABLE soils
                (id INTEGER PRIMARY KEY, date TEXT, lat REAL, lon REAL, vwc INTEGER);''')

# Save (commit) the changes
conn.commit()

```

### 30.4.1 Add Data

In SQLite databases, the construction `(?, ?, ?, ?)` is used as a placeholder for parameter substitution in SQL statements, especially with the `INSERT`, `UPDATE`, and `SELECT` commands. This construction offers the following advantages:

- SQL Injection Prevention:** By using placeholders, you prevent SQL injection, a common web security vulnerability where attackers can interfere with the queries that an application makes to its database.
- Data Handling:** It automatically handles the quoting of strings and escaping of special characters, reducing errors in SQL query syntax due to data.
- Query Efficiency:** When running similar queries multiple times, parameterized queries can improve performance as the database engine can reuse the query plan and execution path.

Each `?` is a placeholder that is replaced with provided data values in a tuple when the `execute` method is called. This ensures that the values are properly formatted and inserted into the database, enhancing security and efficiency.

```

# Insert a row of data
obs = ('2024-01-02', 37.54, -98.78, 38)
cursor.execute("INSERT INTO soils (date, lat, lon, vwc) VALUES (?,?,?,?)", obs)

# Save (commit) the changes
conn.commit()

```

### 30.4.2 Add with multiple entries

You can insert multiple entries at once using `executemany()`

```

# A list of multiple crop records
new_data = [('2024-01-02', 36.54, -98.12, 18),
            ('2024-04-14', 38.46, -99.78, 21),
            ('2024-05-23', 38.35, -98.01, 29)]

# Inserting multiple records at a time
cursor.executemany("INSERT INTO soils (date, lat, lon, vwc) VALUES (?,?,?,?)", new_data)

# Save (commit) the changes
conn.commit()

# Retrieve all data
cursor.execute('SELECT * FROM soils;')
for row in cursor.fetchall():
    print(row)

(1, '2024-01-02', 37.54, -98.78, 38)
(2, '2024-01-02', 36.54, -98.12, 18)
(3, '2024-04-14', 38.46, -99.78, 21)
(4, '2024-05-23', 38.35, -98.01, 29)

```

### 30.4.3 Query data

To query specific data from a table in an SQLite database using the `SELECT` statement, you can specify conditions using the `WHERE` clause. Here's a basic syntax:

```
SELECT col1, col2, ... FROM table_name WHERE condition1 AND condition2;
```

To execute these queries remember to establish a connection, create a cursor object, execute the query using `cursor.execute(query)`, and then use `cursor.fetchall()` to retrieve the results. Close the connection to the database once you're done.

```

# Retrieve all data
cursor.execute('SELECT * FROM soils;')
for row in cursor.fetchall():
    print(row)

(1, '2024-01-02', 37.54, -98.78, 38)
(2, '2024-01-02', 36.54, -98.12, 18)
(3, '2024-04-14', 38.46, -99.78, 21)
(4, '2024-05-23', 38.35, -98.01, 29)

```

```

# Retrieve specific data
cursor.execute('SELECT date FROM soils WHERE vwc >= 30;')
for row in cursor.fetchall():
    print(row)

('2024-01-02',)

# Retrieve specific data (note that we need specify the date as a string)
cursor.execute('SELECT lat,lon FROM soils WHERE date == "2024-03-07";')
for row in cursor.fetchall():
    print(row)

cursor.execute('SELECT * FROM soils WHERE vwc >=30 AND vwc < 40;')
for row in cursor.fetchall():
    print(row)

(1, '2024-01-02', 37.54, -98.78, 38)

```

### 30.4.4 Modify data

You can update records that match certain criteria.

```

# Update quantity for specific date note that this will update both rows with the same date
cursor.execute("UPDATE soils SET vwc = 15 WHERE date = '2024-01-02';")

# Save (commit) the changes
conn.commit()

```

#### 🔥 Update SQLite

Note that the previous command updates both rows with the same date. To only specify a single row we need to be more specific in our command.

```

# Retrieve all data
cursor.execute('SELECT * FROM soils;')
for row in cursor.fetchall():
    print(row)

```

```

(1, '2024-01-02', 37.54, -98.78, 15)
(2, '2024-01-02', 36.54, -98.12, 15)
(3, '2024-04-14', 38.46, -99.78, 21)
(4, '2024-05-23', 38.35, -98.01, 29)

# Update quantity for specific date note that this will update both rows with the same data
cursor.execute("UPDATE soils SET vwc = 5 WHERE date = '2024-01-02' AND id = 2;")

# Save (commit) the changes
conn.commit()

# Retrieve all data
cursor.execute('SELECT * FROM soils;')
for row in cursor.fetchall():
    print(row)

(1, '2024-01-02', 37.54, -98.78, 15)
(2, '2024-01-02', 36.54, -98.12, 5)
(3, '2024-04-14', 38.46, -99.78, 21)
(4, '2024-05-23', 38.35, -98.01, 29)

```

### 30.4.5 Remove data

To remove records, use the DELETE statement.

```

# Delete the Wheat record
cursor.execute("DELETE FROM soils WHERE id = 3")

# Save (commit) the changes
conn.commit()

# Retrieve all data
cursor.execute('SELECT * FROM soils;')
for row in cursor.fetchall():
    print(row)

(1, '2024-01-02', 37.54, -98.78, 15)
(2, '2024-01-02', 36.54, -98.12, 5)
(4, '2024-05-23', 38.35, -98.01, 29)

```

### 30.4.6 Add new column/header

```
# Add a new column
# Use the DEFAULT construction like: DEFAULT 'Unknown'
# to populate new column with custom value
cursor.execute("ALTER TABLE soils ADD COLUMN soil_type TEXT;")

# Retrieve data one more time before we close the database
cursor.execute('SELECT * FROM soils')
for row in cursor.fetchall():
    print(row)

(1, '2024-01-02', 37.54, -98.78, 15, None)
(2, '2024-01-02', 36.54, -98.12, 5, None)
(4, '2024-05-23', 38.35, -98.01, 29, None)
```

### 30.4.7 Closing the connection

Once done with the operations, close the connection to the database.

```
conn.close()
```

## 30.5 Use Pandas to set up database

For this exercise we will use a table of sorghum yields for Franklin county, KS obtained in 2023. The dataset contains breeder brand, hybrid name, yield, moisture, and total weight. The spreadsheet contains metadata on the first line (which we are going to skip) and in the last few rows. From these last rows, we will use functions to match strings and retrieve the planting and harvest dates of the trial. We will then add this information to the dataframe, and then create an SQLite database.

Pandas provides all the methods to read, clean, and export the Dataframe to a SQLite database.

```
# Import modules
import pandas as pd
import sqlite3
```

```

df = pd.read_csv('../datasets/sorghum_franklin_county_2023.csv',
                 skiprows=[0,1,3])
# Inspect first few rows
df.head()

```

|   | BRAND    | NAME    | YIELD | PAVG  | MOIST | TW   |
|---|----------|---------|-------|-------|-------|------|
| 0 | POLANSKY | 5719    | 137.2 | 107.8 | 14.7  | 59.1 |
| 1 | DYNA-GRO | M60GB88 | 135.3 | 106.3 | 14.0  | 57.9 |
| 2 | DYNA-GRO | GX22936 | 134.7 | 105.8 | 13.9  | 58.7 |
| 3 | POLANSKY | 5522    | 132.3 | 103.9 | 13.9  | 58.4 |
| 4 | DYNA-GRO | GX22932 | 131.8 | 103.6 | 14.5  | 59.1 |

```

# Inspect last few rows
df.tail(10)

```

|    | BRAND   | NAME       | YIELD | PAVG  | MOIST | TW   |
|----|---|------------|-------|-------|-------|------|
| 18 | DYNA-GRO  | M67GB87    | 120.5 | 94.7  | 13.9  | 56.1 |
| 19 | DYNA-GRO  | M59GB94    | 117.7 | 92.5  | 13.8  | 57.7 |
| 20 | NaN   | AVERAGE    | 127.2 | 100.0 | 14.1  | 58.3 |
| 21 | NaN   | CV (%)     | 8.4   | 8.4   | 0.3   | 0.8  |
| 22 | NaN   | LSD (0.05) | 5.2   | 4.1   | 0.3   | 0.3  |
| 23 | *Yields must differ by more than the LSD value... | NaN        | NaN   | NaN   | NaN   | NaN  |
| 24 | different.  | NaN        | NaN   | NaN   | NaN   | NaN  |
| 25 | Planted 5-24-23                                   | NaN        | NaN   | NaN   | NaN   | NaN  |
| 26 | Harvested 11-15-23                                | NaN        | NaN   | NaN   | NaN   | NaN  |
| 27 | Fertility 117-38-25-20 Strip till                 | NaN        | NaN   | NaN   | NaN   | NaN  |

```

df.dropna(subset='BRAND', inplace=True)
df.tail(10)

```

|    | BRAND   | NAME    | YIELD | PAVG | MOIST | TW   |
|----|---|---------|-------|------|-------|------|
| 15 | DYNA-GRO  | M63GB78 | 122.7 | 96.4 | 13.9  | 58.0 |
| 16 | DYNA-GRO  | GX22937 | 121.3 | 95.4 | 14.2  | 58.4 |
| 17 | DYNA-GRO  | GX22923 | 121.2 | 95.2 | 13.7  | 55.8 |
| 18 | DYNA-GRO  | M67GB87 | 120.5 | 94.7 | 13.9  | 56.1 |
| 19 | DYNA-GRO  | M59GB94 | 117.7 | 92.5 | 13.8  | 57.7 |
| 23 | *Yields must differ by more than the LSD value... | NaN     | NaN   | NaN  | NaN   | NaN  |

| BRAND                                | NAME | YIELD | PAVG | MOIST | TW  |
|--------------------------------------|------|-------|------|-------|-----|
| 24 different.                        | NaN  | NaN   | NaN  | NaN   | NaN |
| 25 Planted 5-24-23                   | NaN  | NaN   | NaN  | NaN   | NaN |
| 26 Harvested 11-15-23                | NaN  | NaN   | NaN  | NaN   | NaN |
| 27 Fertility 117-38-25-20 Strip till | NaN  | NaN   | NaN  | NaN   | NaN |

```
# Extract planting date
idx = df['BRAND'].str.contains("Planted")
planting_date_str = df.loc[idx, 'BRAND'].values[0]
planting_date = planting_date_str.split(' ')[1]
print(planting_date)
```

5-24-23

```
# Extract harvest date
idx = df['BRAND'].str.contains("Harvested")
harvest_date_str = df.loc[idx, 'BRAND'].values[0]
harvest_date = harvest_date_str.split(' ')[1]
print(harvest_date)
```

11-15-23

```
# Once we are done extracting metadata, let's remove the last few rows
df = df.iloc[:-5]

# Convert header names to lower case to avoid conflict with SQL syntax
df.rename(str.lower, axis='columns', inplace=True)
df.head()
```

|   | brand    | name    | yield | pavg  | moist | tw   | planting_date | harvest_date |
|---|----------|---------|-------|-------|-------|------|---------------|--------------|
| 0 | POLANSKY | 5719    | 137.2 | 107.8 | 14.7  | 59.1 | 5-24-23       | 11-15-23     |
| 1 | DYNA-GRO | M60GB88 | 135.3 | 106.3 | 14.0  | 57.9 | 5-24-23       | 11-15-23     |
| 2 | DYNA-GRO | GX22936 | 134.7 | 105.8 | 13.9  | 58.7 | 5-24-23       | 11-15-23     |
| 3 | POLANSKY | 5522    | 132.3 | 103.9 | 13.9  | 58.4 | 5-24-23       | 11-15-23     |
| 4 | DYNA-GRO | GX22932 | 131.8 | 103.6 | 14.5  | 59.1 | 5-24-23       | 11-15-23     |

```
# Add planting and harvest date to Dataframe to make it more complete
df['planting_date'] = planting_date
df['harvest_date'] = harvest_date
df.head()
```

|   | brand    | name    | yield | pavg  | moist | tw   | planting_date | harvest_date |
|---|----------|---------|-------|-------|-------|------|---------------|--------------|
| 0 | POLANSKY | 5719    | 137.2 | 107.8 | 14.7  | 59.1 | 5-24-23       | 11-15-23     |
| 1 | DYNA-GRO | M60GB88 | 135.3 | 106.3 | 14.0  | 57.9 | 5-24-23       | 11-15-23     |
| 2 | DYNA-GRO | GX22936 | 134.7 | 105.8 | 13.9  | 58.7 | 5-24-23       | 11-15-23     |
| 3 | POLANSKY | 5522    | 132.3 | 103.9 | 13.9  | 58.4 | 5-24-23       | 11-15-23     |
| 4 | DYNA-GRO | GX22932 | 131.8 | 103.6 | 14.5  | 59.1 | 5-24-23       | 11-15-23     |

```
# Use Pandas to turn DataFrame into a SQL Database

# Connect to SQLite database (if it doesn't exist, it will be created)
conn = sqlite3.connect('sorghum_trial.db')

# Write the data to a sqlite table
df.to_sql('sorghum_trial', conn, index=False, if_exists='replace') # to overwrite use option 'replace'

# Close the connection
conn.close()
```

### 30.5.1 Connect, access all data, and close database

```
# Connect to SQLite database (if it doesn't exist, it will be created)
conn = sqlite3.connect('sorghum_trial.db')

# Create cursor
cursor = conn.cursor()

# Access all data
cursor.execute('SELECT * FROM sorghum_trial')
for row in cursor.fetchall():
    print(row)

# Access all data
print('') # Add some white space
cursor.execute('SELECT brand, name FROM sorghum_trial WHERE yield > 130')
```

```

for row in cursor.fetchall():
    print(row)

# Close the connection
conn.close()

('POLANSKY', '5719', 137.2, 107.8, 14.7, 59.1, '5-24-23', '11-15-23')
('DYNA-GRO', 'M60GB88', 135.3, 106.3, 14.0, 57.9, '5-24-23', '11-15-23')
('DYNA-GRO', 'GX22936', 134.7, 105.8, 13.9, 58.7, '5-24-23', '11-15-23')
('POLANSKY', '5522', 132.3, 103.9, 13.9, 58.4, '5-24-23', '11-15-23')
('DYNA-GRO', 'GX22932', 131.8, 103.6, 14.5, 59.1, '5-24-23', '11-15-23')
('DYNA-GRO', 'M72GB71', 130.9, 102.9, 14.5, 59.0, '5-24-23', '11-15-23')
('MATURITY CHECK', 'MED', 128.7, 101.1, 14.0, 58.2, '5-24-23', '11-15-23')
('DYNA-GRO', 'M71GR91', 128.7, 101.1, 14.4, 59.3, '5-24-23', '11-15-23')
('PIONEER', '86920', 128.1, 100.7, 13.9, 57.9, '5-24-23', '11-15-23')
('MATURITY CHECK', 'EARLY', 127.9, 100.5, 14.2, 58.8, '5-24-23', '11-15-23')
('POLANSKY', '5629', 126.5, 99.4, 13.8, 57.2, '5-24-23', '11-15-23')
('MATURITY CHECK', 'LATE', 126.2, 99.2, 14.2, 58.3, '5-24-23', '11-15-23')
('PIONEER', '84980', 125.5, 98.6, 14.1, 58.8, '5-24-23', '11-15-23')
('DYNA-GRO', 'M60GB31', 124.9, 98.2, 14.2, 59.2, '5-24-23', '11-15-23')
('DYNA-GRO', 'GX22934', 122.8, 96.5, 14.6, 59.5, '5-24-23', '11-15-23')
('DYNA-GRO', 'M63GB78', 122.7, 96.4, 13.9, 58.0, '5-24-23', '11-15-23')
('DYNA-GRO', 'GX22937', 121.3, 95.4, 14.2, 58.4, '5-24-23', '11-15-23')
('DYNA-GRO', 'GX22923', 121.2, 95.2, 13.7, 55.8, '5-24-23', '11-15-23')
('DYNA-GRO', 'M67GB87', 120.5, 94.7, 13.9, 56.1, '5-24-23', '11-15-23')
('DYNA-GRO', 'M59GB94', 117.7, 92.5, 13.8, 57.7, '5-24-23', '11-15-23')

('POLANSKY', '5719')
('DYNA-GRO', 'M60GB88')
('DYNA-GRO', 'GX22936')
('POLANSKY', '5522')
('DYNA-GRO', 'GX22932')
('DYNA-GRO', 'M72GB71')

```

# **Part III**

# **EXERCISES**

# 31 Meteogram

A weather meteogram is a time-series graphical representation that displays detailed weather information for a specific location over a continuous period. This compact chart captures various meteorological variables such as air and soil temperature, wind speed and direction, precipitation, cloud cover, and atmospheric pressure. Each variable is typically plotted against time on the horizontal axis, allowing for a clear visual analysis of weather trends and patterns.

```
# Import necessary modules
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Read data and display the first 5 rows
filename = '../datasets/kings_creek_2022_2023_daily.csv'
df = pd.read_csv(filename, parse_dates=['datetime'])
df.head()
```

|   | datetime   | pressure  | tmin  | tmax | tavg   | rmin      | rmax      | prcp | srad    | wspd     | wdir   |
|---|------------|-----------|-------|------|--------|-----------|-----------|------|---------|----------|--------|
| 0 | 2022-01-01 | 96.837917 | -14.8 | -4.4 | -9.60  | 78.475475 | 98.012496 | 0.0  | 2.09808 | 5.483333 | 0.9683 |
| 1 | 2022-01-02 | 97.994583 | -20.4 | -7.2 | -13.80 | 50.543218 | 84.935503 | 0.0  | 9.75636 | 2.216250 | 2.0233 |
| 2 | 2022-01-03 | 97.843750 | -9.4  | 8.8  | -0.30  | 40.622240 | 82.662479 | 0.0  | 9.68076 | 2.749167 | 5.6673 |
| 3 | 2022-01-04 | 96.418750 | 0.1   | 8.6  | 4.35   | 48.326316 | 69.401604 | 0.0  | 8.37900 | 5.805833 | 2.6273 |
| 4 | 2022-01-05 | 97.462500 | -11.1 | -2.2 | -6.65  | 50.341455 | 76.828233 | 0.0  | 5.71716 | 4.206667 | 1.2503 |

```
# Select data for January
start_date = pd.to_datetime('2022-01-01')
end_date = pd.to_datetime('2023-01-01')
idx = (df['datetime'] >= start_date) & (df['datetime'] < end_date)
df = df[idx].copy().reset_index(drop=True)
df.head()
```

|   | datetime   | pressure  | tmin  | tmax | tavg   | rmin      | rmax      | prcp | srad    | wspd     | wdir   |
|---|------------|-----------|-------|------|--------|-----------|-----------|------|---------|----------|--------|
| 0 | 2022-01-01 | 96.837917 | -14.8 | -4.4 | -9.60  | 78.475475 | 98.012496 | 0.0  | 2.09808 | 5.483333 | 0.9688 |
| 1 | 2022-01-02 | 97.994583 | -20.4 | -7.2 | -13.80 | 50.543218 | 84.935503 | 0.0  | 9.75636 | 2.216250 | 2.0233 |
| 2 | 2022-01-03 | 97.843750 | -9.4  | 8.8  | -0.30  | 40.622240 | 82.662479 | 0.0  | 9.68076 | 2.749167 | 5.6673 |
| 3 | 2022-01-04 | 96.418750 | 0.1   | 8.6  | 4.35   | 48.326316 | 69.401604 | 0.0  | 8.37900 | 5.805833 | 2.6273 |
| 4 | 2022-01-05 | 97.462500 | -11.1 | -2.2 | -6.65  | 50.341455 | 76.828233 | 0.0  | 5.71716 | 4.206667 | 1.2500 |

```
# Display number of missing values for each column.
df.isna().sum()
```

```
datetime          0
pressure          0
tmin              0
tmax              0
tavg              0
rmin              0
rmax              0
prcp              0
srad              0
wspd              1
wdir              28
vpd               2
vwc_5cm           0
vwc_20cm          0
vwc_40cm          0
soiltemp_5cm      0
soiltemp_20cm     0
soiltemp_40cm     0
battv             0
dtype: int64
```

```
# Replace missing values
df['wspd'] = df['wspd'].interpolate(method='linear')
df['vpd'] = df['vpd'].interpolate(method='linear')
```

```
# Display the new number of missing values for each column.
df.isna().sum()
```

```
datetime          0
```

```
pressure          0
tmin              0
tmax              0
tavg              0
rmin              0
rmax              0
prcp              0
srad              0
wspd              0
wdir              28
vpd               0
vwc_5cm           0
vwc_20cm          0
vwc_40cm          0
soiltemp_5cm      0
soiltemp_20cm     0
soiltemp_40cm     0
battv             0
dtype: int64
```

## 31.1 Estimate some useful metrics

To characterize what happened during the entire year, let's compute the annual rainfall, maximum wind speed, and maximum and minimum air temperature.

```
# Find and print total precipitation
P_total = df['prcp'].sum().round(2)
print(f'Total precipitation in 2023 was {P_total} mm')
```

```
Total precipitation in 2023 was 523.28 mm
```

```
# Find the total number of days with measurable precipitation
P_hours = (df['prcp'] > 0).sum()
print(f'There were {P_hours} days with precipitation')
```

```
There were 108 days with precipitation
```

```
# Find median air temperature. Print value.  
Tmedian = df['tavg'].median()  
print(f'Median air temperature was {Tmedian} Celsius')
```

Median air temperature was 13.65 Celsius

```
# Find value and time of minimum air temperature. Print value and timestamp.  
fmt = '%A, %B %d, %Y'  
Tmin_idx = df['tmin'].argmin()  
Tmin_value = df.loc[Tmin_idx, 'tmin']  
Tmin_timestamp = df.loc[Tmin_idx, 'datetime']  
print(f'The lowest air temperature was {Tmin_value} on {Tmin_timestamp:{fmt}}')
```

The lowest air temperature was -22.4 on Thursday, December 22, 2022

```
# Find value and time of maximum air temperature. Print value and timestamp.  
Tmax_idx = df['tmax'].argmax()  
Tmax_value = df.loc[Tmax_idx, 'tmax']  
Tmax_timestamp = df.loc[Tmax_idx, 'datetime']  
print(f'The highest air temperature was {Tmax_value} on {Tmax_timestamp:{fmt}}')
```

The highest air temperature was 37.4 on Saturday, July 23, 2022

```
# Find max wind gust and time of occurrence. Print value and timestamp.  
Wmax_idx = df['wspd'].argmax()  
Wmax_value = df.loc[Wmax_idx, 'wspd']  
Wmax_timestamp = df.loc[Wmax_idx, 'datetime']  
print(f'The highest wind speed was {Wmax_value:.2f} m/s on {Wmax_timestamp:{fmt}}')
```

The highest wind speed was 8.15 m/s on Tuesday, April 12, 2022

## 31.2 Meteogram

```
# Create meteogram plot
```

```

# Define style
plt.style.use('ggplot')

# Define fontsize
font = 14

# Create plot
plt.figure(figsize=(14,30))

# Air temperature
plt.subplot(9,1,1)
plt.title('Kings Creek Meteogram for 2022', size=20)
plt.plot(df['datetime'], df['tmin'], color='navy')
plt.plot(df['datetime'], df['tmax'], color='tomato')
plt.ylabel('Air Temperature (°C)', size=font)
plt.yticks(size=font)

# Relative humidity
plt.subplot(9,1,2)
plt.plot(df['datetime'], df['rmin'], color='navy')
plt.plot(df['datetime'], df['rmax'], color='tomato')
plt.ylabel('Relative Humidity (%)', size=font)
plt.yticks(size=font)
plt.ylim(0,100)

# Atmospheric pressure
plt.subplot(9,1,3)
plt.plot(df['datetime'], df['pressure'], '-k')
plt.ylabel('Pressure (kPa)', size=font)
plt.yticks(size=font)

# Vapor pressure deficit
plt.subplot(9,1,4)
plt.plot(df['datetime'], df['vpd'], '-k')
plt.ylabel('Vapor pressure deficit (kPa)', size=font)
plt.yticks(size=font)

# Wind speed
plt.subplot(9,1,5)
plt.plot(df['datetime'], df['wspd'], '-k', label='Wind speed')
plt.ylabel('Wind Speed ($m \ s^{-1}$)', size=font)

```

```

plt.legend(loc='upper left')
plt.yticks(size=font)

# Solar radiation
plt.subplot(9,1,6)
plt.plot(df['datetime'], df['srad'])
plt.ylabel('Solar radiation ($W \ m^{-2}$)', size=font)
plt.yticks(size=font)

# Precipitation
plt.subplot(9,1,7)
plt.step(df['datetime'], df['prcp'], color='navy')
plt.ylabel('Precipitation (mm)', size=font)
plt.yticks(size=font)
plt.ylim(0.01,35)
plt.text(df['datetime'].iloc[5], 30, f"Total = {P_total} mm", size=14)

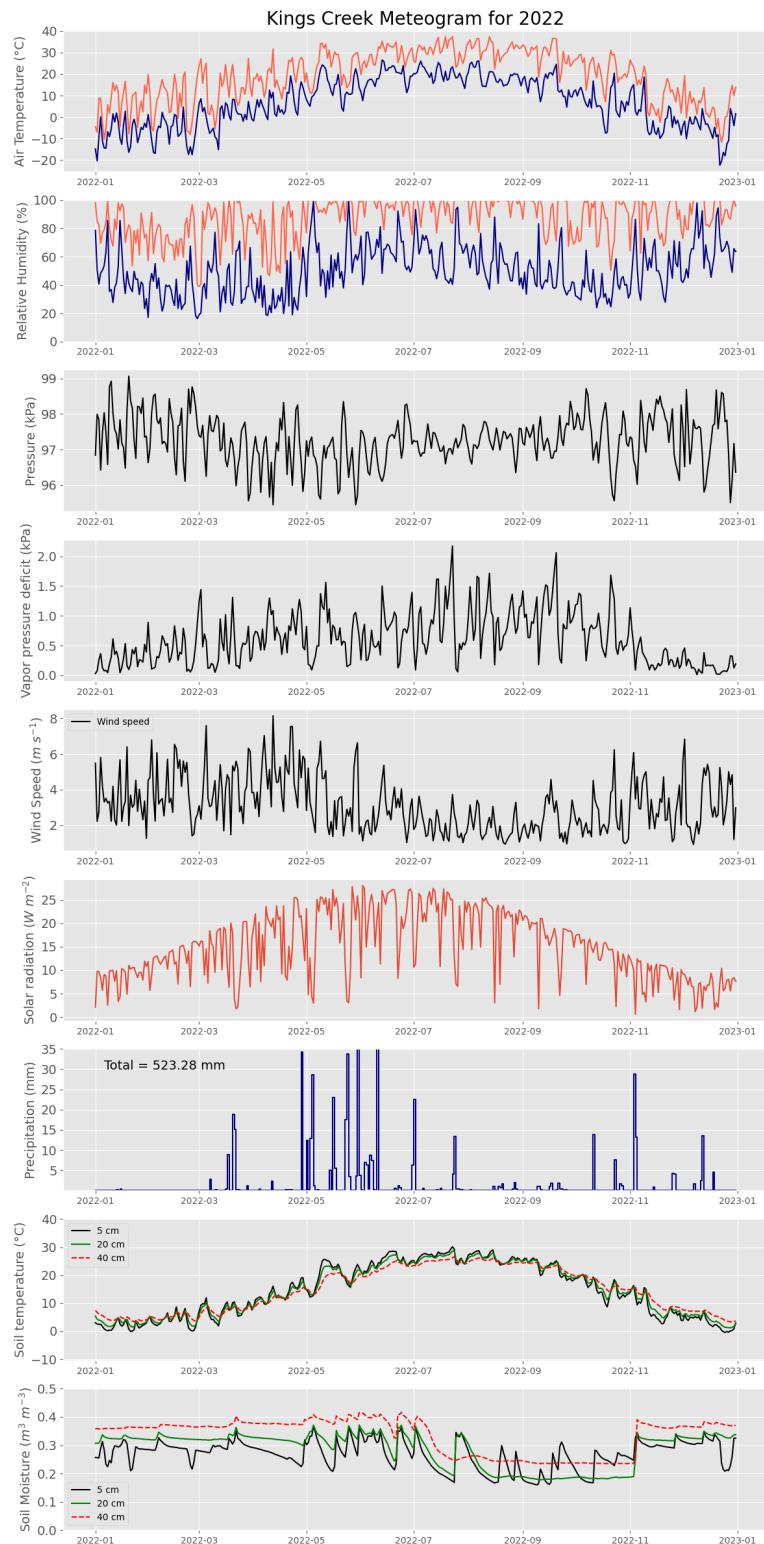
# Soil temperature
plt.subplot(9,1,8)
plt.plot(df['datetime'], df['soiltemp_5cm'], '-k', label='5 cm')
plt.plot(df['datetime'], df['soiltemp_20cm'], '-g', label='20 cm')
plt.plot(df['datetime'], df['soiltemp_40cm'], '--r', label='40 cm')
plt.ylabel('Soil temperature (°C)', size=font)
plt.yticks(size=font)
plt.ylim(df['soiltemp_5cm'].min()-10, df['soiltemp_5cm'].max()+10)
plt.grid(which='minor')
plt.legend(loc='upper left')

# Soil moisture
plt.subplot(9,1,9)
plt.plot(df['datetime'], df['vwc_5cm'], '-k', label='5 cm')
plt.plot(df['datetime'], df['vwc_20cm'], '-g', label='20 cm')
plt.plot(df['datetime'], df['vwc_40cm'], '--r', label='40 cm')
plt.ylabel('Soil Moisture ($m^3 \ m^{-3}$)', size=font)
plt.yticks(size=font)
plt.ylim(0, 0.5)
plt.grid(which='minor')
plt.legend(loc='best')

plt.subplots_adjust(hspace=0.2) # for space between columns wspace=0
# plt.savefig('meteogram.svg', format='svg')

```

```
plt.show()
```



## 32 Group with least variance

Imagine the following scenario: You are planning a greenhouse experiment that consists of growing corn plants to evaluate the response in accumulated biomass after applying to the soil a new nutritional supplement.

For this experiment you would like to start with plants that are as uniform in size as possible (e.g., represented by plant height). So you decide to grow more plants than you need, wait until they have a few leaves fully extended, select the most uniform group of plants among all the plants, and then apply the treatment to start the evaluation of the nutritional supplement.

In this exercise we will use the `itertools` module to find the combination of plants that has the least variance in height.

```
# Import modules
import numpy as np
from itertools import combinations
import matplotlib.pyplot as plt

# Define the number of plants that we need for our experiment
# Say 3 replicates of treated and non-treated plants.
k = 6

# Observations of plant heights (size needs to be larger than k)
obs = np.array([12.0, 12.1, 14.0, 13.5, 14.5, 13.2, 11.8, 14.3, 13.8, 12.9]) # cm

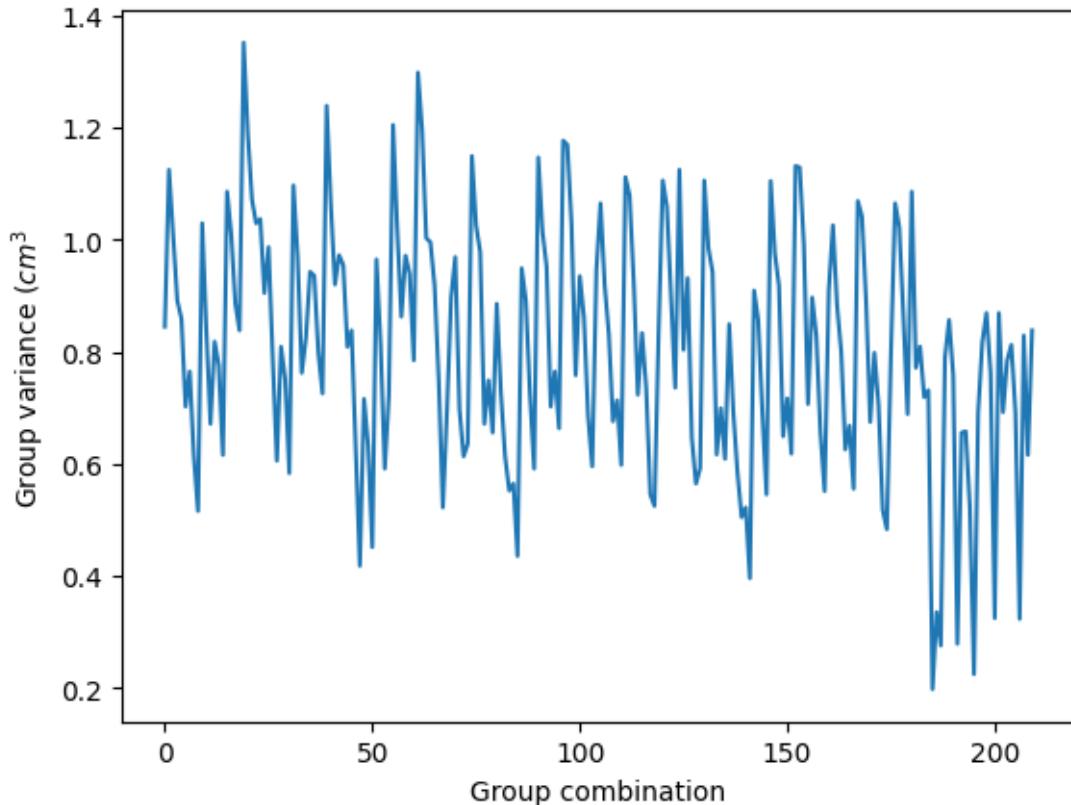
# Generate all possible combinations of k items
groups = list(combinations(range(len(obs)), k))

# Print total number of combinations of k items
print(f'There are a total of {len(groups)} group combinations of {k} items')

# Print first five combinations to inspect the results
print(groups[0:3])
```

There are a total of 210 group combinations of 6 items

```
[(0, 1, 2, 3, 4, 5), (0, 1, 2, 3, 4, 6), (0, 1, 2, 3, 4, 7)]  
  
# Iterate over each group, select values for each member and compute group variance  
  
# Create empty array to append the variance of each group combination  
all_groups_var = np.array([])  
for group in groups:  
  
    # Get group observations  
    # Note that we convert tuple to list to index observations  
    group_obs = obs[list(group)]  
  
    # Compute group variance  
    group_var = np.nanvar(group_obs)  
  
    # Append variance of current group  
    all_groups_var = np.append(all_groups_var, group_var)  
  
  
# Visualize the variance of all groups  
plt.figure()  
plt.plot(all_groups_var)  
plt.xlabel('Group combination')  
plt.ylabel('Group variance ($cm^3$)')  
plt.show()
```



```
# Select group with the lowest variance
idx_min = np.argmin(all_groups_var)
min_var = np.min(all_groups_var)
group_min_var = groups[idx_min]

print(f"Most uniform {k} members are:{group_min_var} with var={min_var:.1f} cm^2")
print(f'The values of combination {idx_min} are: {obs[list(groups[idx_min])]}')
```

Most uniform 6 members are:(2, 3, 4, 5, 7, 8) with var=0.2 cm<sup>2</sup>  
The values of combination 185 are: [14. 13.5 14.5 13.2 14.3 13.8]

### Note

Remember the variance is a metric of dispersion of a set of data points around their mean value, and its units are the square of the units.

## **32.1 Practice**

1. Convert the steps of the script into a function. Make sure to add a docstring and describe the inputs of the function.
2. Can you find and measure some random objects around you and find the most uniform set of  $k$  members of the set? You could measure height, area, volume, mass, or any other property or variable.

## 33 Random Plot Generator

One of the most common statistical operations in research projects is the randomization of treatments. This statistical procedure spans multiple disciplines, from experimental plots in agriculture to Petri dishes in microbiology and it's an inherent step of the scientific method to ensure that overall responses are due to treatments instead of background noise from the environment. The idea is to ensure that differences are only due to treatment effects and not due to an extra factor.

There are multiple experimental designs, but the two simplest and most popular are complete randomized design and randomized complete block design.

**Complete randomized design:** treatments are assigned completely at random so that each experimental unit has the same chance of receiving any one treatment. This design is only useful for homogeneous conditions (Jayaraman, 1984).

In the complete randomized design it should be possible for the same treatment to be more than once within the same replication. Remember here we assume homogeneous conditions and we are not blocking, so any treatment can be assigned to any experimental unit.

**Randomized complete block:** The purpose of blocking is to reduce the experimental error by eliminating the contribution of known sources of variation among the experimental units. For example, terrain slope, temperature gradient in a greenhouse, light conditions, etc. (Jayaraman, 1984).

In the randomized complete block design, each treatment label must be present only once within the block.

The goal of this exercise is to create a Python script that generates random plot labels given the list of treatments and number of replications for both a complete randomized design and a randomized complete block design.

```
# Import modules
import random

# Define treatments
tmt = ["NO", "N25", "N50", "N100", "N200"] # Nitrogen levels in kg of N per hectare
reps = 4
```

### 33.1 Complete Randomized Design

In this case treatments may appear within the same replication more than once. If we are assuming that the background environment (e.g. atmosphere, light conditions, soil) is homogeneous, then there is nothing to worry about having two treatments next to each other in the same replication. We simply want to have several replicates to account for random errors.

```
# Randomized complete block
random.seed('default')
for i in range(0, reps):
    plot_labels = random.sample(tmt, len(tmt))
    print('Replication', i+1, ':', plot_labels)
```

```
Replication 1 : ['N50', 'N200', 'N100', 'N25', 'N0']
Replication 2 : ['N25', 'N200', 'N100', 'N0', 'N50']
Replication 3 : ['N0', 'N25', 'N100', 'N200', 'N50']
Replication 4 : ['N100', 'N200', 'N50', 'N0', 'N25']
```

### 33.2 Complete Randomized Block

Note that in the previous experimental layout the treatments do not repeat within each replication. This is because all treatments need to be present in each block, so that all the treatments are exposed to the same characteristic conditions of each block.

```
# Complete randomized
random.seed('default')

step = len(tmt)
plots = tmt*reps
tmts = random.sample(plots, len(plots))
counter = 0

for i in range(0, len(tmts), step):
    counter += 1
    plot_labels = tmts[i:i+step]
    print('Replication', counter, ':', plot_labels)
```

```
Replication 1 : ['N100', 'N0', 'N50', 'N200', 'N50']
Replication 2 : ['N100', 'N25', 'N0', 'N0', 'N0']
```

```
Replication 3 : ['N25', 'N100', 'N100', 'N50', 'N25']
Replication 4 : ['N200', 'N200', 'N25', 'N50', 'N200']
```

### **33.3 References**

Jayaraman, K., 1984. FORSPA-FAO Publication A Statistical Manual for forestry Research (No. Fe 25). FAO,. <http://www.fao.org/3/X6831E/X6831E07.htm>

## 34 Particle Random Walk

A random walk is an important concept in physics and chemistry that describes [Brownian motion](#). In this challenge you are required to simulate and track the path of a single particle, ignoring collisions with the boundaries (although this could be a nice addition).

The goal is to write a Python script that simulates the random movement of a particle (e.g. a colloid in aqueous suspension). This is often called a random walk since for each time step the particle will randomly “walk” from its current location to a new location. In this challenge you will need to create a figure to visualize the path of the particle.

```
# Import modules
import numpy as np
from bokeh.plotting import figure, output_notebook, show
output_notebook()
TOOLS = "pan,box_zoom,reset,save"
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_1

```
# Set seed for reproducible results (optional)
np.random.seed(10)

# Initial particle position
current_xpos = 0
current_ypos = 0

# Initial list of positions
x = [current_xpos]
y = [current_ypos]

# Define number of particle steps
N = 1000
```

```

# Generate set of random steps in advance (it could also be done inside the for loop)
xstep = np.random.randint(-1,2,N)
ystep = np.random.randint(-1,2,N)

# Iterate and track the particle over each step
for i in range(N):

    # Update position
    current_xpos += xstep[i]
    current_ypos += ystep[i]

    # Append new position
    x.append(current_xpos)
    y.append(current_ypos)

# Plot random walk
f = figure(plot_width=600, plot_height=400)
f.line(x=x, y=y)
f.circle(0, 0, size=15, color='green')
f.circle(current_xpos, current_ypos, size=15, color='red')

show(f)

```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_e

### 34.1 Solution without for loop

```

# Set random seed for reproducibility
np.random.seed(10)

# Number of particle steps
N = 1000

# Generate set of random steps
xstep = np.random.randint(-1,2,N)
ystep = np.random.randint(-1,2,N)

```

```

# Cumulative sum (cumulative effect) of random choices
x = xstep.cumsum()
y = ystep.cumsum()

# For completeness add the initial position
# Omitting this step will not cause any noticeable difference in the plot
x = np.insert(x,0,0)
y = np.insert(y,0,0)

# Generate plot of particle path
f = figure(plot_width=600, plot_height=400)
f.line(x=x, y=y)
f.circle(0, 0, size=15, color='green')
f.circle(current_xpos, current_ypos, size=15, color='red')

show(f)

```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_e

## 34.2 Shortest solution

```

np.random.seed(10)
N = 1000
x = np.random.randint(-1,2,N).cumsum()
y = np.random.randint(-1,2,N).cumsum()

# Plot random walk
f = figure(plot_width=600, plot_height=400)
f.line(x=x, y=y)
show(f)

```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_e

## 35 Runoff

The curve number method is a simple empirical method to approximate the amount of runoff from rainfall events. The method was developed by the US Department of Agriculture Soil Conservation Service based on observed runoff values from field experiments.

Curve numbers range from 0 to 100, where higher numbers represent more runoff. For instance, a completely impervious surface like a paved parking lot or a house roof would receive a curve number of 100. The curve number of agricultural fields typically ranges from 60 to 90, depending on the slope, vegetation cover, tillage, and soil physical properties associated to water infiltration and hydraulic conductivity.

For instance, a field planted with a small grain crop (e.g., wheat, oats, barley) that has abundant surface stubble, presence of terraces, and was planted following elevation contours on a well-structured soil would have a curve number of about 60.

Visit [this link](#) to learn more about the curve number method.

```
# Import modules
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Define cure number function
def curve_number(P, CN=75):
    """
    Curve number method proposed by the Soil Conservation Service
    P is precipitation in milimeters
    CN is the curve number
    """
    runoff = np.zeros_like(P)

    S02 = 1000/CN - 10;
    S005 = 1.33 * S02**1.15;
    Lambda = 0.05; # Hawkins, 2002.
    Ia = S005 * Lambda; # Initial abstraction (Ia). Rainfall before runoff starts to occur
    idx = P > Ia
```

```

runoff[idx] = (P[idx] - Ia)**2 / (P[idx] - Ia + S005);

return runoff

# Test function
curve_number(np.array([2]),100)

array([2])

# Load data from Acme, OK
df = pd.read_csv('../datasets/acme_ok_daily.csv')
df['Date'] = pd.to_datetime(df['Date'], format='%m/%d/%y %H:%M')
df['RAIN'].fillna(0)
df.head()

```

|   | Date       | DOY | TMAX      | TMIN      | RAIN  | HMAX | HMIN  | ATOT | W2AVG    | ETgrass  |
|---|------------|-----|-----------|-----------|-------|------|-------|------|----------|----------|
| 0 | 2005-01-01 | 1   | 21.161111 | 14.272222 | 0.00  | 97.5 | 65.97 | 4.09 | 5.194592 | 1.976940 |
| 1 | 2005-01-02 | 2   | 21.261111 | 4.794444  | 0.00  | 99.3 | 77.37 | 4.11 | 3.428788 | 1.302427 |
| 2 | 2005-01-03 | 3   | 5.855556  | 3.477778  | 2.54  | 99.8 | 98.20 | 2.98 | 3.249973 | 0.349413 |
| 3 | 2005-01-04 | 4   | 4.644444  | 0.883333  | 7.62  | 99.6 | 98.50 | 1.21 | 3.527137 | 0.288802 |
| 4 | 2005-01-05 | 5   | 0.827778  | -9.172222 | 24.13 | 99.4 | 86.80 | 1.65 | NaN      | 0.367956 |

```

# Select a specific year to study runoff
idx_year = df['Date'].dt.year == 2007
df = df[idx_year].reset_index(drop=True) # drop=True prevents adding the old index to the
df.head()

```

|   | Date       | DOY | TMAX      | TMIN      | RAIN  | HMAX | HMIN  | ATOT  | W2AVG    | ETgrass  |
|---|------------|-----|-----------|-----------|-------|------|-------|-------|----------|----------|
| 0 | 2007-01-01 | 1   | 9.111111  | -4.533333 | 0.000 | 91.0 | 23.25 | 12.93 | 2.096612 | 1.602857 |
| 1 | 2007-01-02 | 2   | 10.122222 | -6.900000 | 0.000 | 95.2 | 25.79 | 11.35 | 0.965604 | 1.106730 |
| 2 | 2007-01-03 | 3   | 12.322222 | 0.394444  | 0.000 | 80.1 | 45.66 | 8.40  | 3.741716 | 2.007247 |
| 3 | 2007-01-04 | 4   | 9.922222  | 5.388889  | 2.032 | 97.8 | 68.65 | 3.68  | 3.504785 | 1.047221 |
| 4 | 2007-01-05 | 5   | 12.450000 | 5.194444  | 0.000 | 98.2 | 68.39 | 6.15  | 3.862416 | 1.238152 |

```

# Compute cumulative rainfall
df['RAIN_SUM'] = df['RAIN'].cumsum()

```

```

# Compute cumulative runoff
df['RUNOFF'] = curve_number(df['RAIN']/25.4,CN=80)*25.4
df['RUNOFF_SUM'] = df['RUNOFF'].cumsum()

# Check Dataframe
df.head()

```

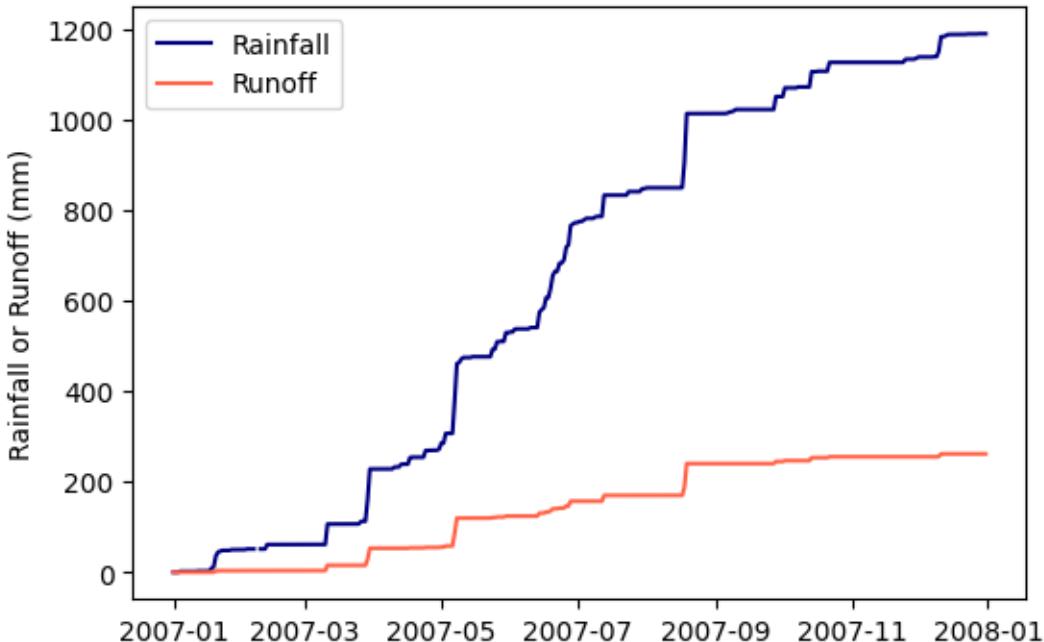
|   | Date       | DOY | TMAX      | TMIN      | RAIN  | HMAX | HMIN  | ATOT  | W2AVG    | ETgrass  | RA   |
|---|------------|-----|-----------|-----------|-------|------|-------|-------|----------|----------|------|
| 0 | 2007-01-01 | 1   | 9.111111  | -4.533333 | 0.000 | 91.0 | 23.25 | 12.93 | 2.096612 | 1.602857 | 0.00 |
| 1 | 2007-01-02 | 2   | 10.122222 | -6.900000 | 0.000 | 95.2 | 25.79 | 11.35 | 0.965604 | 1.106730 | 0.00 |
| 2 | 2007-01-03 | 3   | 12.322222 | 0.394444  | 0.000 | 80.1 | 45.66 | 8.40  | 3.741716 | 2.007247 | 0.00 |
| 3 | 2007-01-04 | 4   | 9.922222  | 5.388889  | 2.032 | 97.8 | 68.65 | 3.68  | 3.504785 | 1.047221 | 2.03 |
| 4 | 2007-01-05 | 5   | 12.450000 | 5.194444  | 0.000 | 98.2 | 68.39 | 6.15  | 3.862416 | 1.238152 | 2.03 |

```

# Plot cumulative rainfall and runoff

plt.figure(figsize=(6,4))
plt.plot(df['Date'], df['RAIN_SUM'], color='navy', label='Rainfall')
plt.plot(df['Date'], df['RUNOFF_SUM'], color='tomato', label='Runoff')
plt.ylabel('Rainfall or Runoff (mm)')
plt.legend()
plt.show()

```



```
print('Annual precipitation = ', df['RAIN_SUM'].iloc[-1].round(), ' mm')
print('Annual runoff = ', df['RUNOFF_SUM'].iloc[-1].round(), ' mm')
```

Annual precipitation = 1190.0 mm  
 Annual runoff = 262.0 mm

## 35.1 Practice

- Using precipitation observations for 2007, what is the total runoff for a fallow under bare soil for a soil with hydrologic condition D?
- Select a year in which the total runoff is lower than for 2007. Use a curve number of 80.
- Modify the curve number function so that it works with precipitation data in both inches and millimeters.

## 35.2 References

Ponce, V.M. and Hawkins, R.H., 1996. Runoff curve number: Has it reached maturity?. Journal of hydrologic engineering, 1(1), pp.11-19.

# 36 Mixing problems

Mixing problems in the context of a tank containing a solute, are a classic example used to demonstrate principles in differential equations. These problems typically involve a tank filled with a liquid into which a solute (like a salt or chemical) is introduced, either continuously or at specific intervals. The challenge is to determine the concentration of the solute in the tank over time, considering factors such as the rate of solute addition, the volume of the liquid in the tank, and the mixing or outflow rates. These scenarios are not only academically interesting but also have practical applications in industries like agronomy and environmental engineering.

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
```

## 36.1 Example 1: Tank level and salt concentration

A 1500 gallon tank initially contains 600 gallons of water with 5 lbs of salt dissolved in it. Water enters the tank at a rate of 9 gal/hr and the water entering the tank has a salt concentration of  $\frac{1}{5}(1 + \cos(t))$  lbs/gal. If a well-mixed solution leaves the tank at a rate of 6 gal/hr:

- how long does it take for the tank to overflow?
- how much salt (total amount in lbs) is in the entire tank when it overflows?

Assume each iteration is equivalent to one hour

We will break the problem into two steps. The first step consists of focusing on the tank volume and leaving the calculation of the salt concentration aside. Trying to solve both questions at the same time can make this problem more difficult than it actual is. A good way of thinking about this problem is by making an analogy with the balance of a checking account (tank level), where we have credits (inflow rate, salary) and debits (outflow, expenses).

### 36.1.1 Step 1: Find time it takes to fill the tank

```
# Initial parameters
tank_capacity = 1500 # gallons
tank_level = 600      # gallons Initial tank_level
inflow_rate = 9        # gal/hr
outflow_rate = 6        # gal/hr

# Step 1: Compute tank volume and determine when the tank is full
counter_hours = 0
while tank_level < tank_capacity:
    tank_level = tank_level + inflow_rate - outflow_rate
    counter_hours += 1

print('Hours:', counter_hours)
print('Tank level:', tank_level)
```

```
Hours: 300
Tank level: 1500
```

### 36.1.2 Step 2: Add the calculation of the amount of salt

Now that we understand the problem in simple terms and we were able to implement it in Python, is time to add the computation of salt concentration at each time step. In this step is important to realize that concentration is amount of salt per unit volume of water, in this case gallons of water. Following the same reasoning of the previous step, we now need to calculate the balance of salt taking into account initial salt content, inflow, and outflow. So, to solve the problem we need:

- the inflow rate of water with salt
- the salt concentration of the inflow rate
- the outflow rate of water with salt
- the salt concentration of the outflow rate (**we need to calculate this**)

From the statement we have the first 3 pieces of information, but we lack the last one. Since concentration is mass of salt per unit volume of water, we just need to divide the total amount of salt over the current volume of water in the tank. So at the beginning we have 5 lbs/600 gallons = 0.0083 lbs/gal, which will be the salt concentration of the outflow during the first hour. Because the amount of water and salt in the tank changes every hour, we need to include this computation in each iteration to update the salt concentration of the outflow.

```

# Initial parameters
period = 1000          # Large number of hours to ensure hours
tank_level = np.ones(period)*np.nan # Pre-allocate array with NaNs
salt_mass = np.ones(period)*np.nan # Pre-allocate array with NaNs
tank_level[0] = 600    # gallons
salt_mass[0] = 5       # lbs
tank_capacity = 1500 # gallons
inflow_rate = 9        # gal/hr
outflow_rate = 6       # gal/hr

# Compute tank volume and salt mass at time t until tank is full
for t in range(1,period):

    # The salt concentration will be computed using the tank level of the previous hour
    salt_inflow = 1/5*(1+np.cos(t)) * inflow_rate # lbs/gal ranges between 0 and 0.4
    salt_outflow = salt_mass[t-1]/tank_level[t-1] * outflow_rate
    salt_mass[t] = salt_mass[t-1] + salt_inflow - salt_outflow

    # Now we can update the tank level
    tank_level[t] = tank_level[t-1] + inflow_rate - outflow_rate # volume of the tank

    # Added <greater than> just in case the tank level does not exactly match the tank capacity
    # For instance, if we set the condition to '==' and the tank_level changes from 1499 to 1500
    # between two iteration steps, then the loop will never stop.
    if tank_level[t] >= tank_capacity:
        print(t, 'hours')
        print(np.round(salt_mass[t]),'lbs of salt')
        break

```

300 hours  
280.0 lbs of salt

```

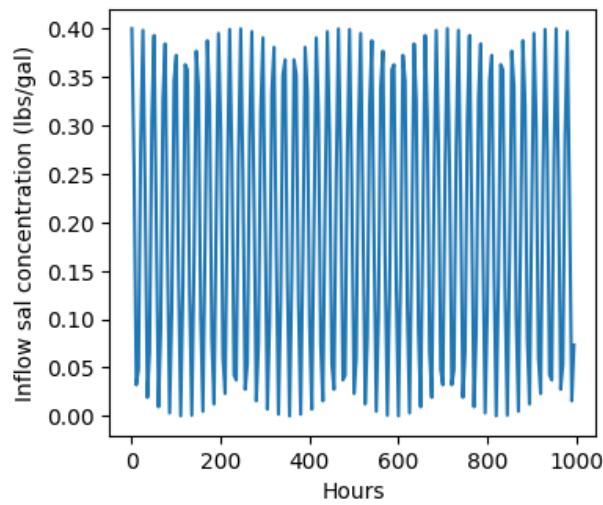
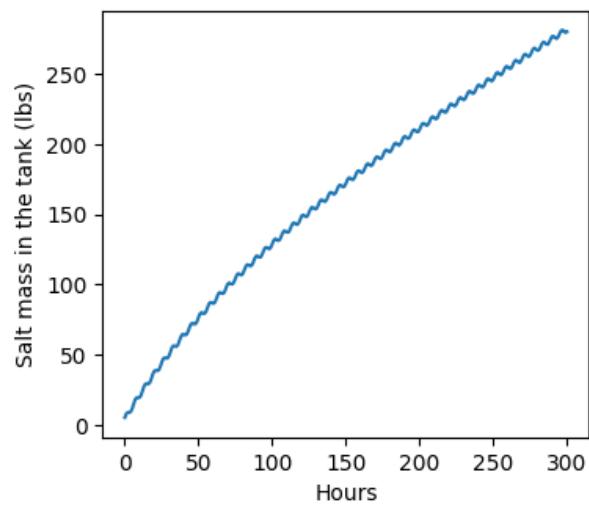
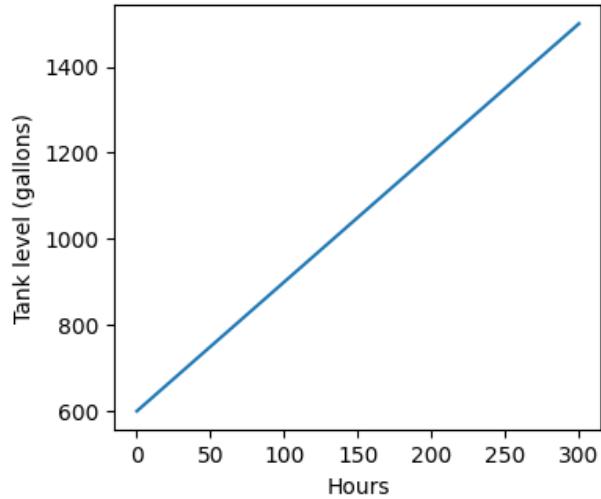
# Create figures
plt.figure(figsize=(4,12))

plt.subplot(3,1,1)
plt.plot(range(period),tank_level)
plt.xlabel('Hours')
plt.ylabel('Tank level (gallons)')

```

```
plt.subplot(3,1,2)
plt.plot(range(period),salt_mass)
plt.xlabel('Hours')
plt.ylabel('Salt mass in the tank (lbs)')

plt.subplot(3,1,3)
# Plot every 5 values to clearly see the curve in the figure
plt.plot(range(0,period,5), 1/5*(1+np.cos(range(0,period,5))))
plt.xlabel('Hours')
plt.ylabel('Inflow sal concentration (lbs/gal)')
plt.show()
```



### 36.1.3 Example 2: Excess of herbicide problem

A farmer is preparing to control weeds in a field crop using a sprayer with a tank containing 100 liters of fresh water. The recommended herbicide concentration to control the weeds without affecting the crop is 2%, which means that the farmer would need to add 2 liters of herbicide to the tank. However, due to an error while measuring the herbicide, the farmer adds 3 liters of herbicide instead of 2 liters, which will certainly kill the weeds, but may also damage the crop and contaminate the soil with unnecessary product. To fix the problem and avoid disposing the entire tank, the farmer decides to open the outflow valve to let some herbicide solution out of the tank at a rate of 3 liters per minute, while at the same time, adding fresh water from the top of the tank at a rate of 3 liters per minute. Assume that the tank has a stirrer that keeps the solution well-mixed (i.e. the herbicide concentration at any given time is homogeneous across the tank).

Indicate the time in minutes at which the herbicide concentration in the tank is restored at 2% (0.02 liters of herbicide per liter of fresh water). In other words, you need to find the time at which the farmer needs to close the outflow valve.

```
# Numerical Solution
tank_level = 100 # Liters
chemical_volume = 3 # Liters
chemical_concentration = chemical_volume/tank_level # Liter of chemical per Liter of water
inflow_rate = 3 # Liters per minute
outflow_rate = 3 # Liters per minute
recommended_concentration = 0.02 # Liter of chemical per Liter of water
dt = 0.1 # Time of each iteration in minutes
counter = 0 # Time tracker in minutes

while chemical_concentration > recommended_concentration:
    tank_level = tank_level + inflow_rate*dt - outflow_rate*dt
    chemical_inflow = 0
    chemical_outflow = chemical_volume/tank_level*outflow_rate*dt
    chemical_volume = chemical_volume + chemical_inflow - chemical_outflow
    chemical_concentration = chemical_volume/tank_level
    counter += dt

print('Solution:',round(counter,1),'minutes')
```

Solution: 13.5 minutes

## 36.2 References

The examples in this notebook were adapted from problems and exercises in Morris. Tenenbaum and Pollard, H., 1963. Ordinary differential equations: an elementary textbook for students of mathematics, engineering, and the sciences. Dover Publications.

# 37 Mass-volume relationships

Mass-volume relationships are fundamental variables for understanding and describing the soil composition in terms of the mass and volume of the main soil constituents (solids, water, air). Key variables include bulk density, which measures the mass of soil per unit volume, and porosity, indicating the volume fraction of soil that can be occupied by air and water. These relationships are crucial for assessing soil health, estimating soil fertility, and water storage. Understanding mass-volume relationships is vital for effective soil management, influencing practices like irrigation, tillage, and crop selection, ultimately impacting agricultural productivity.

## 37.1 Problem 1

A cylindrical soil sample with a diameter of 5.00 cm and a height of 5.00 cm has a wet mass of 180.0 g. After oven-drying the soil sample at 105 degrees Celsius for 48 hours, the sample has a dry mass of 147.0 g. Based on the provided information, calculate:

- volume of the soil sample:  $V_t = \pi r^2 h$
- mass of water in the sample:  $M_{water} = M_{wet\ soil} - M_{dry\ soil}$
- bulk density:  $\rho_b = M_{dry\ soil}/V_t$
- porosity:  $f = 1 - \rho_b/\rho_p$
- gravimetric water content:  $\theta_g = M_{water}/M_{dry\ soil}$
- volumetric water content:  $\theta_v = V_{water}/V_t$
- relative saturation:  $\theta_{rel} = \theta_v/f$
- soil water storage expressed in mm and inches of water:  $S = \theta_v * z$

Throughout the exercise, assume a particle density of 2.65 g cm<sup>-3</sup> and that water has a density of 0.998 g cm<sup>-3</sup>

```
# Import modules
import numpy as np

# Problem information
sample_diameter = 5.00 # cm
sample_height = 5.00 # cm
```

```

wet_mass = 180.0 # grams
dry_mass = 147.0 # grams
density_water = 0.998 # g/cm^3 at 20 Celsius
particle_density = 2.65 # g/cm^3

# Sample volume
sample_volume = np.pi * (sample_diameter/2)**2 * sample_height
print(f'Volume of sample is: {sample_volume:.0f} cm^3')

```

Volume of sample is: 98 cm<sup>3</sup>

```

# Mass of water in the sample
mass_water = wet_mass - dry_mass
print(f'Mass of water is: {mass_water:.0f} g')

```

Mass of water is: 33 g

```

# Bulk density
bulk_density = dry_mass/sample_volume
print('Bulk density of the sample is:', round(bulk_density,2), 'g/cm^3')

```

Bulk density of the sample is: 1.5 g/cm<sup>3</sup>

```

# Porosity
f = (1 - bulk_density/particle_density)
print(f'Porosity of the sample is: {f:.2f}') # Second `f` is for floating-point

```

Porosity of the sample is: 0.43

```

# Gravimetric soil moisture
# Mass of water per unit mass of dry soil. Typically in g/g or kg/kg
theta_g = mass_water / dry_mass
print(f'Gravimetric water content is: {theta_g:.3f} g/g')

```

Gravimetric water content is: 0.224 g/g

```

# Volumetric soil moisture
# Volume of water per unit volume of dry soil. Typically in cm^3/cm^3 or m^3/m^3
volume_water = mass_water / density_water
theta_v = volume_water / sample_volume
print(f'Volumetric water content is: {theta_v:.3f} cm^3/cm^3')

```

Volumetric water content is: 0.337 cm<sup>3</sup>/cm<sup>3</sup>

```

# Relative saturation
rel_sat = theta_v/f
print(f'Relative saturation is: {rel_sat:.2f}')

```

Relative saturation is: 0.77

```

# Storage
storage_mm = theta_v * sample_height*10 # convert from cm to mm
storage_in = storage_mm/25.4 # 1 inch = 25.4 mm
print(f'The soil water storage in mm is: {storage_mm:.1f} mm')
print(f'The soil water storage in inches is: {storage_in:.3f} inches')

```

The soil water storage in mm is: 16.8 mm  
The soil water storage in inches is: 0.663 inches

## 37.2 Problem 2

How many liters of water are stored in the top 1 meter of the soil profile of a field that has an area of 64 hectares (about 160 acres)? Assume that average soil moisture of the field is the volumetric water content computed in the previous problem.

### **i** Note

Note Recall that the volume ratio is the same as the length ratio. This means that we can use the volumetric water content to represent the ‘height of water’ in the field. If you are having trouble visualizing this fact, grab a cylindrical container and fill it with 25% of its volume in water. Then measure the height of the water relative to the height of the container, it should also be 25%. Note that this will not work for conical shapes, so make sure to grab a uniform shape like a cylinder or a cube.

```

# Liters of water in a field
field_area = 64*10_000 # 1 hectare = 10,000 m^2
profile_length = 1 # meter
equivalent_height_of_water = profile_length * theta_v # m of water
volume_of_water = field_area * equivalent_height_of_water # m^3 of water

# Use the fact that 1 m^3 = 1,000 liters
liters_of_water = volume_of_water * 1_000
print(f'There are {round(liters_of_water)} liters of water')

# Compare volume of water to an Olympic-size swimming pool (50 m x 25 m x 2 m)
pool_volume = 50 * 25 * 2 # m^3
print(f'This is equivalent to {round(liters_of_water/pool_volume)} olympic swimming pools!')

```

There are 215557669 liters of water  
This is equivalent to 86223 olympic swimming pools!

### 37.3 Problem 3

Imagine that we want to change the soil texture of this field in the rootzone (top 1 m). How much sand, silt, or clay do we need to haul in to change the textural composition by say 1%? While different soil fractions usually have slightly different bulk densities, let's use the value from the previous problem. We are not looking for the exact number, we just want a ballpark idea. So, what is 1% of the total mass of the field considering the top 1 m of the soil profile?

```

# Field area was already in converted from hectares to m^2
field_volume = field_area * 1 # m^3

# Since 1 Mg/m^3 = g/cm^3, we use the same value (1 Megagram = 1 metric ton)
field_mass = field_volume * bulk_density

one_percent_mass = field_mass/100
print(f'1% of the entire field mass is {one_percent_mass:.1f} Mg')

ultra_truck_maxload = 450 # Mg or metric tons
number_of_truck_loads_required = one_percent_mass / ultra_truck_maxload
print(f'It would require the load of {number_of_truck_loads_required:.0f} trucks')

```

1% of the entire field mass is 9582.9 Mg  
It would require the load of 21 trucks

## 38 Soil textural classes

The United States Department of Agriculture (USDA) Natural Resources Conservation Service (NRCS) has identified a soil classification system that consists of 12 soil textural classes based on the relative composition of sand, silt, and clay particles. The proportion of sand, silt, and clay content determines the water retention, drainage, aeration, and nutrient-holding capacity of a soil.

The Soil Textural Triangle is a tool commonly used to visually represent these classifications. These classes include sandy clay, silty clay, clay, sandy clay loam, clay loam, silty clay loam, sandy loam, loam, silt loam, silt, loamy sand, and sand.

**Sand:** These are the coarsest particles, with a diameter ranging from 0.05 to 2.0 millimeters. Sand particles are gritty when rubbed between fingers and do not form a ball when moistened. Sand contributes to good drainage and aeration of the soil but has a low capacity to hold nutrients and water.

**Silt:** Silt particles are finer than sand, with a diameter ranging from 0.002 to 0.05 millimeters. Silt feels smooth when dry and silky when wet. It has a greater ability than sand to hold water and nutrients, but is less aerated than sand.

**Clay:** These are the finest soil particles, with a diameter of less than 0.002 millimeters. Clay is sticky and plastic when wet and can be rolled into thin ribbons. When dry, it becomes very hard and can forms well-aggregated and cohesive clods. Clay has a high nutrient and water-holding capacity but excessive clay contents often lead to problems associated with poor infiltration, drainage, and aeration.

In this exercise we will write code to automate the determination of the soil textural class of any given soil based on the percent of sand and clay content. Since this code will have lots of fixed ranges and rules, a function seems to be the right approach, so that we do this heavy and tedious task only once and then we re-use the code as needed.

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt

def find_soil_textural_class(sand,clay):
    """
```

Function that returns the USDA-NRCS soil textural class given the percent sand and clay.

Parameters:

sand (float, integer): Sand content as a percentage

clay (float, integer): Clay content as a percentage

Returns:

string: One of the 12 soil textural classes

Authors:

Andres Patrignani

Date created:

12 Jan 2024

Source:

E. Benham and R.J. Ahrens, W.D. 2009.

Clarification of Soil Texture Class Boundaries.

Nettleton National Soil Survey Center, USDA-NRCS, Lincoln, Nebraska.

"""

```
if not isinstance(sand, (int, float, np.int64)):
    raise TypeError(f"Input type {type(sand)} is not valid.")

try:
    # Determine silt content
    silt = 100 - sand - clay

    if sand + clay > 100:
        raise Exception('Inputs add over 100%')
    elif sand < 0 or clay < 0:
        raise Exception('One or more inputs are negative')

except ValueError as e:
    return f"Invalid input: {e}"

# Classification rules
if silt + 1.5*clay < 15:
    textural_class = 'sand'
```

```

    elif silt + 1.5*clay >= 15 and silt + 2*clay < 30:
        textural_class = 'loamy sand'

    elif (clay >= 7 and clay < 20 and sand > 52 and silt + 2*clay >= 30) or (clay < 7 and
        textural_class = 'sandy loam'

    elif clay >= 7 and clay < 27 and silt >= 28 and silt < 50 and sand <= 52:
        textural_class = 'loam'

    elif (silt >= 50 and clay >= 12 and clay < 27) or (silt >= 50 and silt < 80 and clay <
        textural_class = 'silt loam'

    elif silt >= 80 and clay < 12:
        textural_class = 'silt'

    elif clay >= 20 and clay < 35 and silt < 28 and sand > 45:
        textural_class = 'sandy clay loam'

    elif clay >= 27 and clay < 40 and sand > 20 and sand <= 45:
        textural_class = 'clay loam'

    elif clay >= 27 and clay < 40 and sand <= 20:
        textural_class = 'silty clay loam'

    elif clay >= 35 and sand > 45:
        textural_class = 'sandy clay'

    elif clay >= 40 and silt >= 40:
        textural_class = 'silty clay'

    elif clay >= 40 and sand <= 45 and silt < 40:
        textural_class = 'clay'

    else:
        textural_class = 'unknown' # in case we failed to catch any errors earlier

    return textural_class

# Calculate soil texture for a trivial case
sand = 100
clay = 0

```

```

find_soil_textural_class(sand,clay)

'sand'

# Let's create some random soils
N = 50
np.random.seed(1)
clay_list = np.random.randint(0, 60, N) # Hard to find soils with >60% clay
sand_list = np.random.randint(0, 100-clay_list, N) # Ensure sum is <= 100%
silt_list = 100 - clay_list - sand_list

# Inspect list
clay_list

array([37, 43, 12, 8, 9, 11, 5, 15, 0, 16, 1, 12, 7, 45, 6, 25, 50,
       20, 37, 18, 20, 11, 42, 28, 29, 14, 50, 4, 23, 23, 41, 49, 55, 30,
       32, 22, 13, 41, 9, 7, 22, 57, 1, 0, 17, 8, 24, 13, 51, 47])

# Classify the soils using the function we just created
soil_textural_class = []
for sand,clay in zip(sand_list,clay_list):
    current_class = find_soil_textural_class(sand,clay)

from matplotlib.ticker import AutoMinorLocator, MultipleLocator
from matplotlib._cm import _Set3_data
from mpltern.datasets import soil_texture_classes

# Install mpltern library using the following command:
# !pip install mpltern

# THe code below was obtained directly from the following website (mpltern docs):
# Source: https://mpltern.readthedocs.io/en/latest/gallery/miscellaneous/soil\_texture.html
# All teh credit for this wonderful library goes to the creators of the mpltern library

def calculate_centroid(vertices):
    """Calculte the centroid of a polygon.

https://en.wikipedia.org/wiki/Centroid#Of\_a\_polygon

```

```

Parameters
-----
vertices : (n, 2) np.ndarray
    Vertices of a polygon.

Returns
-----
centroid : (2, ) np.ndarray
    Centroid of the polygon.

"""
roll0 = np.roll(vertices, 0, axis=0)
roll1 = np.roll(vertices, 1, axis=0)
cross = np.cross(roll0, roll1)
area = 0.5 * np.sum(cross)
return np.sum((roll0 + roll1) * cross[:, None], axis=0) / (6.0 * area)

def plot_soil_texture_classes(ax):
    """Plot soil texture classes."""
    classes = soil_texture_classes

    for (key, value), color in zip(classes.items(), _Set3_data):
        tn0, tn1, tn2 = np.array(value).T
        patch = ax.fill(tn0, tn1, tn2, ec="k", fc=color, alpha=0.6, zorder=2.1)
        centroid = calculate_centroid(patch[0].get_xy())

        # last space replaced with line break
        label = key[::-1].replace(" ", "\n", 1)[::-1].capitalize()

        ax.text(centroid[0], centroid[1], label,
                ha="center", va="center", transform=ax.transData,
                )

    ax.xaxis.set_major_locator(MultipleLocator(10.0))
    ax.xaxis.set_minor_locator(AutoMinorLocator(2))
    ax.yaxis.set_major_locator(MultipleLocator(10.0))
    ax.yaxis.set_minor_locator(AutoMinorLocator(2))

```

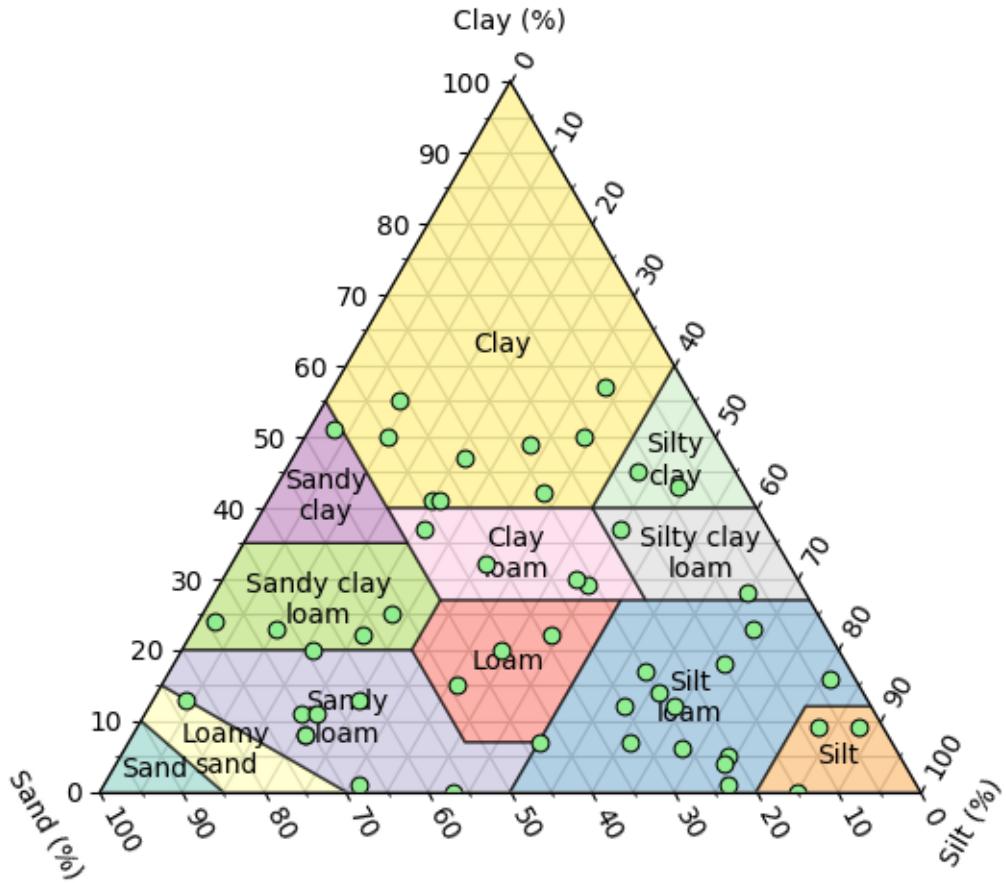
```
ax.grid(which="both")

ax.set_tlabel("Clay (%)")
ax.set_llabel("Sand (%)")
ax.set_rlabel("Silt (%)")

ax.taxis.set_ticks_position("tick2")
ax.laxis.set_ticks_position("tick2")
ax.raxis.set_ticks_position("tick2")

# Create triangle with our random samples overlaid

ax = plt.subplot(projection="ternary", ternary_sum=100.0)
plot_soil_texture_classes(ax)
ax.scatter(clay_list, sand_list, silt_list, zorder=3, marker='o',
           s=40, facecolor='lightgreen', edgecolor='k', linewidth=0.75)
plt.show()
```



## 38.1 References

- Benham E., Ahrens, R.J., and Nettleton, W.D. (2009). Clarification of Soil Texture Class Boundaries. Nettleton National Soil Survey Center, USDA-NRCS, Lincoln, Nebraska.
- Yuji Ikeda. (2023). yuzie007/mpltern: 1.0.2 (1.0.2). Zenodo. <https://doi.org/10.5281/zenodo.8289090>

# 39 Distribution daily precipitation

The size of daily rainfall events is a relevant variable in agronomy, hydrology, and forestry. Small rainfall events are often intercepted by plant canopies and litter on the soil surface, so they are not very effective for increasing soil moisture.

In this tutorial we will explored a long-term (1980-2020) dataset for Greeley county in western Kansas to create a histogram and fit a probability density function to records of daily rainfall events.

```
# Import modules
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
```

## 39.1 Read and prepare dataset for analysis

```
# Load data
filename = '../datasets/Greeley_Kansas.csv'
df = pd.read_csv(filename, parse_dates=['timestamp'])

# Check first few rows
df.head(3)
```

|   | id       | longitude   | latitude  | timestamp  | doy | pr       | rmax       | rmin      | sph      | sr |
|---|----------|-------------|-----------|------------|-----|----------|------------|-----------|----------|----|
| 0 | 19800101 | -101.805968 | 38.480534 | 1980-01-01 | 1   | 2.942802 | 89.670753  | 54.058212 | 0.002494 | 7. |
| 1 | 19800102 | -101.805968 | 38.480534 | 1980-01-02 | 2   | 0.446815 | 100.000000 | 52.695320 | 0.003245 | 5. |
| 2 | 19800103 | -101.805968 | 38.480534 | 1980-01-03 | 3   | 0.000000 | 100.000000 | 65.851830 | 0.002681 | 9. |

```
# Add year column, so that we can group events and totals by year
df.insert(1, 'year', df['timestamp'].dt.year)
df.head(3)
```

|   | id       | year | longitude   | latitude  | timestamp  | doy | pr       | rmax       | rmin      | sph    |
|---|----------|------|-------------|-----------|------------|-----|----------|------------|-----------|--------|
| 0 | 19800101 | 1980 | -101.805968 | 38.480534 | 1980-01-01 | 1   | 2.942802 | 89.670753  | 54.058212 | 0.0024 |
| 1 | 19800102 | 1980 | -101.805968 | 38.480534 | 1980-01-02 | 2   | 0.446815 | 100.000000 | 52.695320 | 0.0032 |
| 2 | 19800103 | 1980 | -101.805968 | 38.480534 | 1980-01-03 | 3   | 0.000000 | 100.000000 | 65.851830 | 0.0020 |

## 39.2 Find value and date of largest daily rainfall event on record

To add some context, let's find out what is the largest daily rainfall event in the period 1980-2020 for Greeley county, KS.

```
# Find largest rainfall event and the date
amount_largest_event = df['pr'].max()
idx_largest_event = df['pr'].argmax()
date_largest_event = df.loc[idx_largest_event, 'timestamp']

print(f'The largest rainfall was {amount_largest_event:.1f} mm')
print(f'and occurred on {date_largest_event:%Y-%m-%d}')
```

The largest rainfall was 68.6 mm  
and occurred on 2010-05-18

## 39.3 Probability density function of precipitation amount

The most important step before creating the histogram is to identify days with daily rainfall greater than 0 mm. If we don't do this, we will include a large number of zero occurrences, that will affect the distribution. We know that it does not rain every day in this region, but when it does, what is the typical size of a rainfall event?

```
# Boolean to identify days with rainfall greater than 0 mm
idx_rained = df['pr'] > 0

# For brevity we will create a new variable with all the precipitation events >0 mm
data = df.loc[idx_rained,'pr']

# Determine median daily rainfall (not considering days without rain)
median_rainfall = data.median()
print(f'Median daily rainfall is {median_rainfall:.1f} mm')
```

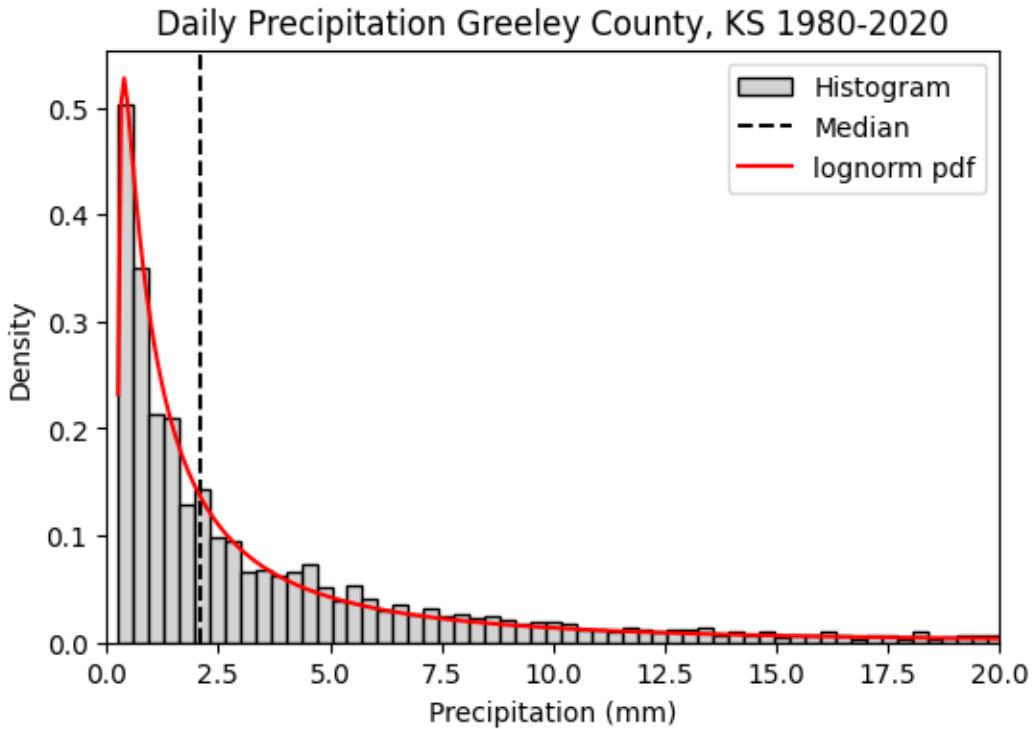
Median daily rainfall is 2.1 mm

```
# Fit theoretical distribution function
# I assumed a lognormal distribution based on the shape of the histogram
# and the fact that rainfall cannot be negative
bounds = [(1, 10),(0.1,10),(0,10)] # Guess bounds for `s` parameters of the lognorm pdf
fitted_pdf = stats.fit(stats.lognorm, data, bounds)
print(fitted_pdf.params)
```

```
FitParams(s=1.5673979274845327, loc=0.23667991099824695, scale=1.6451037468195546)
```

```
# Create vector from 0 to x_max to plot the lognorm pdf
x = np.linspace(data.min(), data.max(), num=1000)

# Create figure
plt.figure(figsize=(6,4))
plt.title('Daily Precipitation Greeley County, KS 1980-2020')
plt.hist(data, bins=200, density=True,
          facecolor='lightgrey', edgecolor='k', label='Histogram')
plt.axvline(median_rainfall, linestyle='--', color='k', label='Median')
plt.plot(x, stats.lognorm.pdf(x, *fitted_pdf.params), color='r', label='lognorm pdf')
plt.xlabel('Precipitation (mm)')
plt.ylabel('Density')
plt.xlim([0, 20])
plt.legend()
plt.show()
```



## 39.4 Cumulative density function

We can also ask: What is the probability of having a daily rainfall event equal or lower than  $x$  amount? The empirical cumulative distribution can help us answer this question. Note that if we ask greater than  $x$  amount, we will need to use the complementary cumulative distribution function (basically  $1-p$ ).

```
# Select rainfall events lower or equal than a specific amount
amount = 5

# Use actual data to estimate the probability
p = stats.lognorm.cdf(amount, *fitted_pdf.params)
print(f'The probability of having a rainfall event <= {amount} mm is {p:.2f}')
print(f'The probability of having a rainfall event > {amount} mm is {1-p:.2f}'')
```

The probability of having a rainfall event  $\leq 5$  mm is 0.75  
The probability of having a rainfall event  $> 5$  mm is 0.25

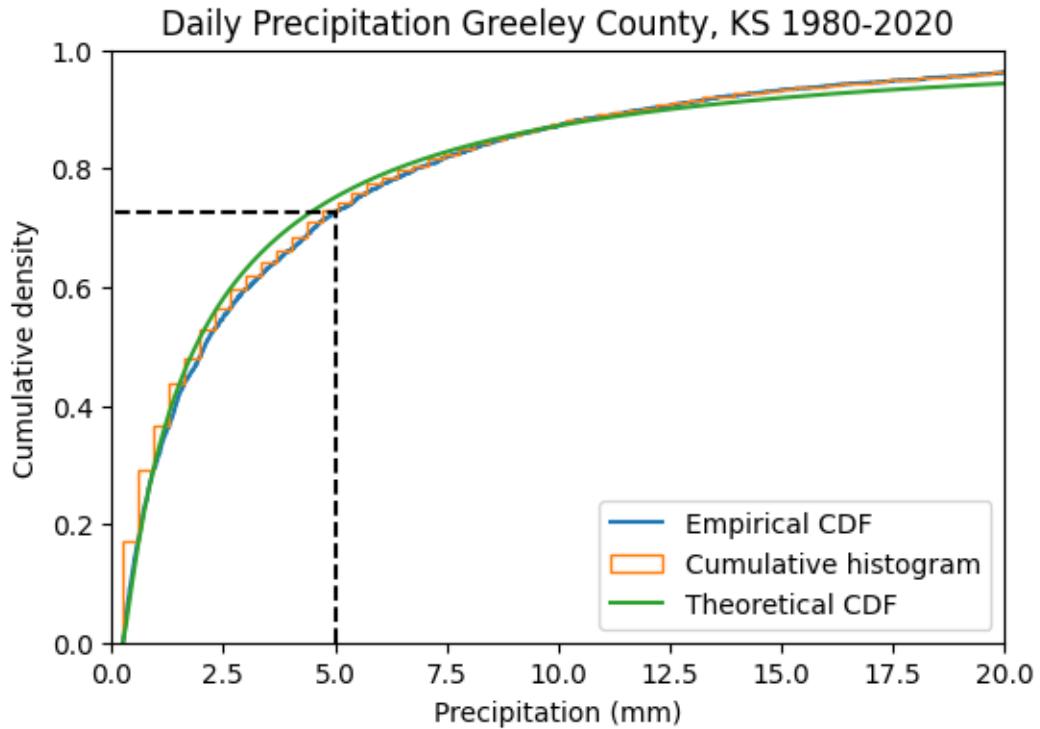
### Note

To determine the probability using the observations, rather than the fitted theoretical distribution, we can simply use the number of favorable cases over the total number of possibilities, like this:

```
idx_threshold = data <= amount
p = np.sum(idx_threshold) / np.sum(idx_rained)
```

```
# Cumulative distributions
plt.figure(figsize=(6,4))
plt.ecdf(data, label="Empirical CDF")
plt.hist(data, bins=200, density=True, histtype="step",
         cumulative=True, label="Cumulative histogram")
plt.plot(x, stats.lognorm.cdf(x, *fitted_pdf.params), label='Theoretical CDF')

plt.plot([amount, amount, 0],[0, p, p], linestyle='--', color='k')
plt.title('Daily Precipitation Greeley County, KS 1980-2020')
plt.xlabel('Precipitation (mm)')
plt.ylabel('Cumulative density')
plt.xlim([0, 20])
plt.legend()
plt.show()
```



In this tutorial we learned that: - typical daily rainfall events in places like western Kansas tend to be very small, in the order of 2 mm. We found this by inspecting a histogram and computing the median of all days with measurable rainfall.

- the probability of having rainfall events larger than 5 mm is only 25%. We found this by using a cumulative density function. If we consider that rainfall interception by plant canopies and crop residue can range from 1 to several millimeters, daily rainfall events will be ineffective reaching the soil surface in regions with this type of daily rainfall distributions, where most of the soil water recharge will depend on larger rainfall events.

## 39.5 References

Clark, O. R. (1940). Interception of rainfall by prairie grasses, weeds, and certain crop plants. Ecological monographs, 10(2), 243-277.

Dunkerley, D. (2000). Measuring interception loss and canopy storage in dryland vegetation: a brief review and evaluation of available research strategies. Hydrological Processes, 14(4), 669-678.

Kang, Y., Wang, Q. G., Liu, H. J., & Liu, P. S. (2004). Winter wheat canopy-interception with its influence factors under sprinkler irrigation. In 2004 ASAE Annual Meeting (p. 1). American Society of Agricultural and Biological Engineers.

## 40 High-resolution rainfall events

Rainfall is often reported using daily totals, like 20 mm per day or 1 inch per day. For some applications like flash flooding prediction and the design of urban drainage systems, hourly or even 15-minute rainfall measurements are often more meaningful for accurate forecasting and process modeling. But what happens when we record rainfall at even higher temporal resolutions (e.g., 1-minute, 5-minute)?

To investigate individual rainfall events, we will use a dataset recorded at 1-minute intervals at the Kansas State University Rocky Ford Experiment Station, near Manhattan, KS during July 2020. Rainfall observations were recorded using a tipping bucket rain gauge with an 9.6-inch funnel opening and a precision of 0.1 mm per tipping (model TE525MM, Texas Electronics, Inc.). This dataset has 44,638 values (60 minutes per hour x 24 hours per day x 31 days). It would take slightly over 122 years of daily rainfall data to reach the same number of records presented here for just one month.

To identify individual rainfall events from 1-minute interval observations, we will use a for loop to iterate over each record. During this process, a state variable will track whether we are inside or outside of a rainfall event as follows: if the recorded rainfall for the current minute is greater than 0 mm, this indicates the occurrence of a rainfall event. Conversely, a reading of zero rainfall does not immediately signal the end of an event. This is because of the possibility of light rain that may not activate the gauge's tipping mechanism in that specific minute. To address this intra-event intermittency, we will use an “inter-event time” criterion that defines a specific duration of continuous zero rainfall measurements, which, when met, confirms the conclusion of the ongoing rainfall event. I added a few articles in the References section in case you want to dive deeper into this topic.

```
# Import modules
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from pprint import pprint

# Load rainfall data
df = pd.read_csv('../datasets/rocky_ford_july_2020_1min.csv',
                 skiprows=[1,2], na_values='NAN',
```

```

        parse_dates=['TIMESTAMP'], date_format='%Y-%m-%d %H:%M:%S')

# Show length of dataframe
print(df.shape)

# Display a few rows to inspect the nature of the dataset
df.head(3)

```

(44638, 2)

|   | TIMESTAMP           | Rain_mm |
|---|---------------------|---------|
| 0 | 2020-07-01 00:01:00 | 0.0     |
| 1 | 2020-07-01 00:02:00 | 0.0     |
| 2 | 2020-07-01 00:03:00 | 0.0     |

```

# Identify individual rainfall events

# Create empty list to append summary metrics for each rainfall event
D = []

# Settings
within_event = False # Boolean to denote whether iteration is within a rainfall event or not
interval = 1 # Measurement interval of the raw data (1-minute in our study)
inter_event_time = 60 # Minimum inter-event time (MIT) in interval units (minutes in our study)

# Initialize variables
counter_no_rainfall = 0 # Track time without rainfall events
event_rain_time = 0 # Time during the event with measurable rainfall
event_rain_amount = 0 # Total rainfall amount per event in millimeters
event_max_intensity = 0 # Maximum rainfall intensity (converted to mm/hr)

# Iterate over each measurements
for k, row in df.iterrows():

    # Get current rainfall amount
    rain = row['Rain_mm']

    # If rainfall is greater than zero that means we are within a rainfall event...
    if rain > 0:

```

```

# ...so we proceed to reset the counter
counter_no_rainfall = 0

# Set parameters if this the first value greater than zero rainfall
if not within_event:

    # Convert to True because now we are inside a new event
    within_event = True

    # Rainfall is totaled at the end of the minute, so we grab the previous minute
    idx_start = k - 1

    # Track duration, amount, and intensity of the precipitation event
    event_rain_time += interval
    event_rain_amount += rain

    # Update maximum interval rainfall rate
    event_max_intensity = max(event_max_intensity, rain)

else:
    # Track number of intervals without rainfall
    counter_no_rainfall += interval

# End of the rainfall event if there is a certain user-specified cumulative period with
if within_event and (counter_no_rainfall == inter_event_time):

    # Get timestamp right before first rainfall for this event
    event_start_time = df["TIMESTAMP"].iloc[idx_start]

    # Get timestamp of last recorded rainfall for this event
    idx_end = k - inter_event_time
    event_end_time = df['TIMESTAMP'].iloc[idx_end]

    # Compute total duration of the event (including short periods without rain)
    event_duration = (event_end_time - event_start_time).total_seconds()/60 + 1

    # Append dictionary to list of events
    D.append({'event_start_time':event_start_time,
              'event_end_time':event_end_time,
              'event_rain_time':event_rain_time,
              'event_duration':event_duration,

```

```

        'event_amount':round(event_rain_amount,1),
        'event_max_intensity':event_max_intensity*60/interval, # converted to mm
        'event_timestamps': df.loc[idx_start:idx_end, 'TIMESTAMP'].values,
        'event_rain': df.loc[idx_start:idx_end, 'Rain_mm'].values}
    )

# Reset variables before starting new iteration
within_event = False
event_rain_time = 0
event_rain_amount = 0
event_max_intensity = 0

# Print to let us know the code is working
print(f'Saved event {event_start_time} - {event_end_time}')

```

```

Saved event 2020-07-03 01:24:00 - 2020-07-03 01:57:00
Saved event 2020-07-03 03:26:00 - 2020-07-03 03:27:00
Saved event 2020-07-03 05:04:00 - 2020-07-03 05:47:00
Saved event 2020-07-05 16:21:00 - 2020-07-05 16:26:00
Saved event 2020-07-07 05:45:00 - 2020-07-07 05:46:00
Saved event 2020-07-09 01:55:00 - 2020-07-09 05:30:00
Saved event 2020-07-09 06:49:00 - 2020-07-09 06:50:00
Saved event 2020-07-14 06:48:00 - 2020-07-14 07:24:00
Saved event 2020-07-15 03:47:00 - 2020-07-15 10:09:00
Saved event 2020-07-18 04:15:00 - 2020-07-18 04:22:00
Saved event 2020-07-18 07:40:00 - 2020-07-18 07:41:00
Saved event 2020-07-19 05:54:00 - 2020-07-19 05:55:00
Saved event 2020-07-20 06:03:00 - 2020-07-20 07:34:00
Saved event 2020-07-20 11:09:00 - 2020-07-20 11:10:00
Saved event 2020-07-20 23:01:00 - 2020-07-21 01:12:00
Saved event 2020-07-21 05:03:00 - 2020-07-21 08:32:00
Saved event 2020-07-21 11:02:00 - 2020-07-21 11:33:00
Saved event 2020-07-24 10:50:00 - 2020-07-24 10:51:00
Saved event 2020-07-26 21:52:00 - 2020-07-27 00:12:00
Saved event 2020-07-27 01:30:00 - 2020-07-27 01:31:00
Saved event 2020-07-27 09:34:00 - 2020-07-27 11:04:00
Saved event 2020-07-27 12:15:00 - 2020-07-27 12:16:00
Saved event 2020-07-27 20:06:00 - 2020-07-27 20:07:00
Saved event 2020-07-29 06:32:00 - 2020-07-29 08:38:00
Saved event 2020-07-29 13:03:00 - 2020-07-29 14:07:00

```

```
Saved event 2020-07-29 15:56:00 - 2020-07-29 16:48:00
Saved event 2020-07-30 04:11:00 - 2020-07-30 04:37:00
Saved event 2020-07-30 10:02:00 - 2020-07-30 10:59:00
Saved event 2020-07-30 15:58:00 - 2020-07-30 16:45:00
Saved event 2020-07-30 18:22:00 - 2020-07-30 20:29:00
```

```
# Explore one of the events
pprint(D[3])
```

```
{'event_amount': 0.2,
 'event_duration': 6.0,
 'event_end_time': Timestamp('2020-07-05 16:26:00'),
 'event_max_intensity': 6.0,
 'event_rain': array([0. , 0.1, 0. , 0. , 0. , 0.1]),
 'event_rain_time': 2,
 'event_start_time': Timestamp('2020-07-05 16:21:00'),
 'event_timestamps': array(['2020-07-05T16:21:00.000000000', '2020-07-05T16:22:00.000000000',
                           '2020-07-05T16:23:00.000000000', '2020-07-05T16:24:00.000000000',
                           '2020-07-05T16:25:00.000000000', '2020-07-05T16:26:00.000000000'],
                           dtype='datetime64[ns]')}
```

```
# Conver resulting dictionary into a new Dataframe for easier data exploration
df_events = pd.DataFrame(D)
df_events.head(3)
```

|   | event_start_time    | event_end_time      | event_rain_time | event_duration | event_amount | event_m |
|---|---------------------|---------------------|-----------------|----------------|--------------|---------|
| 0 | 2020-07-03 01:24:00 | 2020-07-03 01:57:00 | 29              | 34.0           | 5.9          | 30.0    |
| 1 | 2020-07-03 03:26:00 | 2020-07-03 03:27:00 | 1               | 2.0            | 0.1          | 6.0     |
| 2 | 2020-07-03 05:04:00 | 2020-07-03 05:47:00 | 11              | 44.0           | 2.7          | 42.0    |

```
# Find event with largest duration
idx_max_duration = df_events['event_duration'].argmax()
event_max_duration = df_events.loc[idx_max_duration, 'event_duration']

print(f'Maximum duration: {event_max_duration} minutes. Event #{idx_max_duration}')
```

```
Maximum duration: 383.0 minutes. Event #8
```

```

# Find event with largest duration
idx_max_amount = df_events['event_amount'].argmax()
event_max_amount = df_events.loc[idx_max_amount, 'event_amount']

print(f'Maximum amount: {event_max_amount} mm. Event #{idx_max_amount}')

Maximum amount: 41.3 mm. Event #12

# Find event with maximum rainfall intensity
# Find event with largest duration
idx_max_intensity = df_events['event_max_intensity'].argmax()
event_max_intensity = df_events.loc[idx_max_intensity, 'event_max_intensity']

print(f'Maximum intensity: {event_max_intensity} mm/hr. Event #{idx_max_intensity}')

```

Maximum intensity: 198.0 mm/hr. Event #12

### **i** Note

The event numbers above are for the Pandas Dataframe, which means that they are zero-index. Keep this in mind, particularly if you save the Dataframe as a spreadsheet. One option to avoid confusion is to also save the index (e.g., not using the `index=False`, which is the default anyways)

```

# Select event
event = 12

# Generate quick plot
locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
fmt = mdates.ConciseDateFormatter(locator)

# Re-define for more compact code and easy figure text insertion
x = df_events.loc[event, 'event_timestamps']
y = df_events.loc[event, 'event_rain'].cumsum()

plt.figure()
plt.plot(x, y)
plt.xlabel('Time')
plt.ylabel('Cumulative precipitation (mm)')

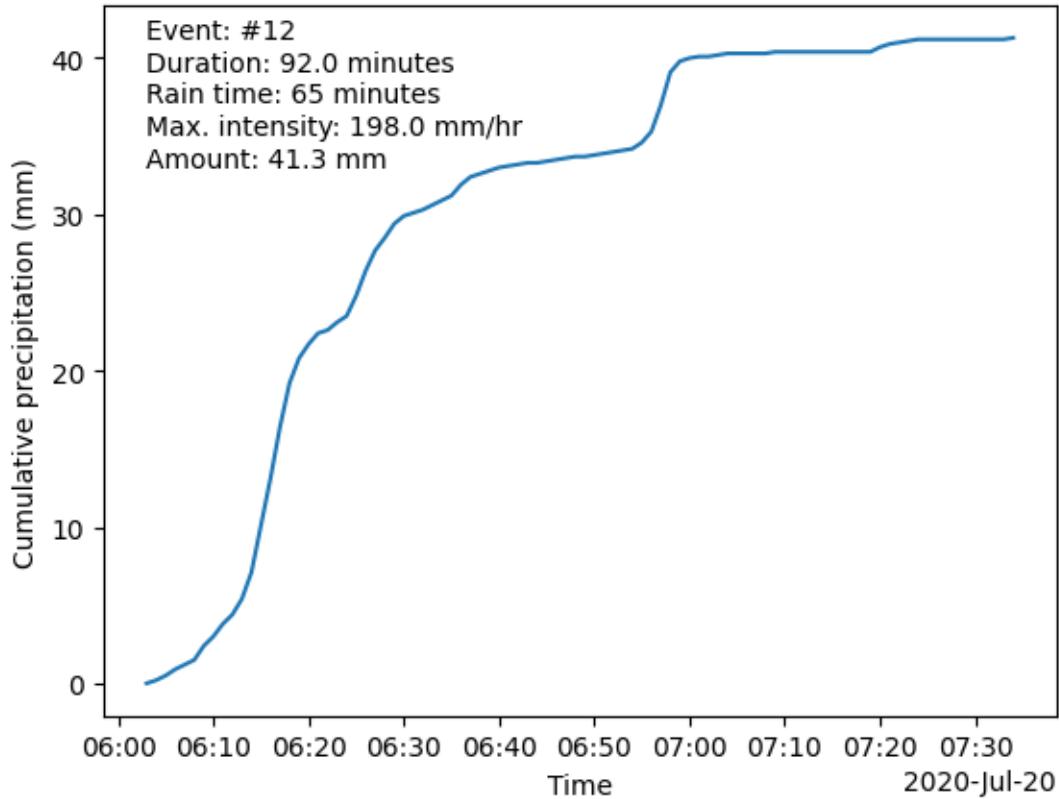
```

```

plt.gca().xaxis.set_major_formatter(fmt)
plt.text(x[0], y[-1], f"Event: #{event}")
plt.text(x[0], y[-1]*0.95, f"Duration: {df_events.loc[event, 'event_duration']} minutes")
plt.text(x[0], y[-1]*0.9, f"Rain time: {df_events.loc[event, 'event_rain_time']} minutes")
plt.text(x[0], y[-1]*0.85, f"Max. intensity: {df_events.loc[event, 'event_max_intensity']}")
plt.text(x[0], y[-1]*0.8, f"Amount: {df_events.loc[event, 'event_amount']} mm")

plt.show()

```



```

# Plot all the events together
plt.figure()
for k,row in df_events.iterrows():
    x = np.arange(row['event_duration'])
    y = row['event_rain'].cumsum()

    # Generate plot

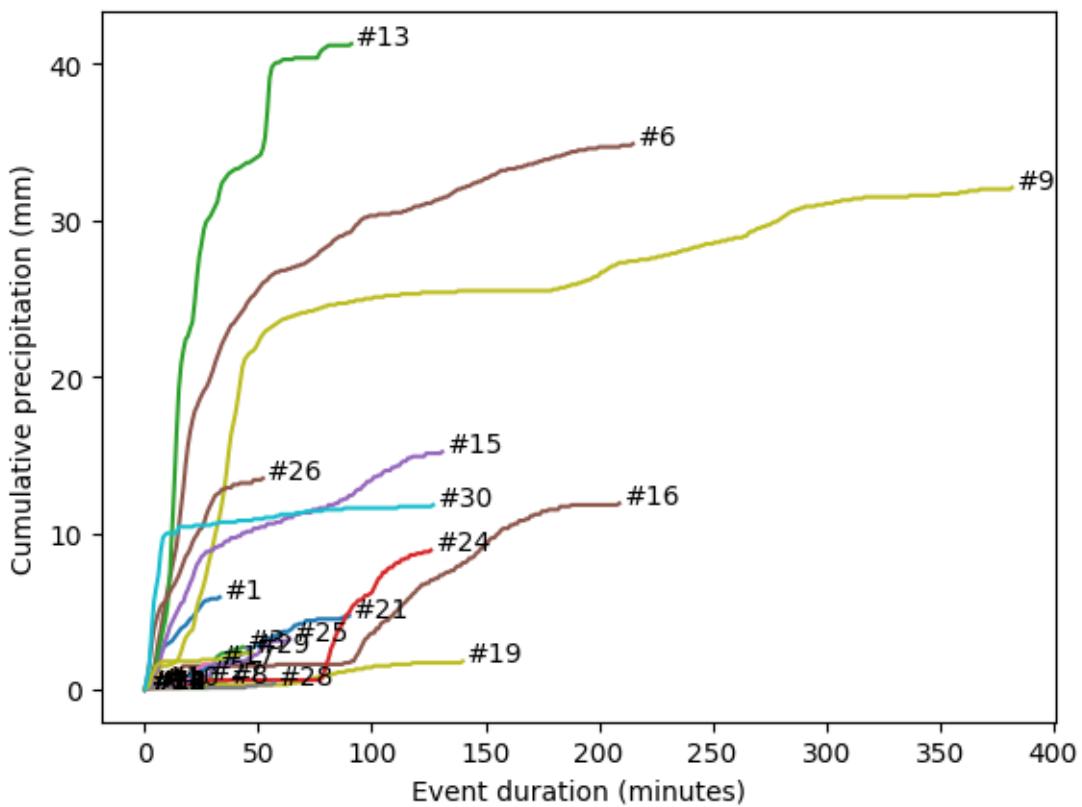
```

```

plt.plot(x, y)
plt.text(x[-1]+2, y[-1], f'#{k+1}') # Added 1 to match a possible published table

plt.xlabel('Event duration (minutes)')
plt.ylabel('Cumulative precipitation (mm)')
plt.show()

```



## 40.1 References

Dunkerley, D. (2015). Intra-event intermittency of rainfall: An analysis of the metrics of rain and no-rain periods. *Hydrological Processes*, 29(15), 3294-3305.

Dyer, D. W., Patrignani, A., & Bremer, D. (2022). Measuring turfgrass canopy interception and throughfall using co-located pluviometers. *Plos one*, 17(9), e0271236. <https://doi.org/10.1371/journal.pone.0271236>

# 41 First and last frost

The concept of first and last frost dates is essential in the context of gardening and agriculture, as it helps in determining the suitable planting and harvesting times. A frost date typically refers to when temperatures are expected to fall to the threshold level that can cause damage to plants. This is generally at or below 0°C. It's important to note that these frost dates are not absolute but are probabilistic estimates based on historical data.

In addition to the first and last frost dates, the concept of the “frost-free window” or “growing season” refers to the period between the last spring frost and the first fall frost, during which temperatures are generally above the freezing point.

The length of the frost-free window varies depending on the geographical location and local climatic conditions. For instance, in warmer regions, this period may be quite long, allowing for an extended growing season suitable for a wide variety of crops. In contrast, colder regions may have a shorter frost-free window, limiting the types of plants that can be grown.

In this exercise we will use a long-term dataset from 1980 to 2020 to estimate the:

- typical date for first and last frost
- frost-free period
- probability density and cumulative density functions

```
# Import modules
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats

# Read data
df = pd.read_csv('../datasets/Johnson_Kansas.csv', parse_dates=['timestamp'])
df.head(3)
```

|   | id       | longitude  | latitude  | timestamp  | doy | pr  | rmax      | rmin      | sph      | srad     |
|---|----------|------------|-----------|------------|-----|-----|-----------|-----------|----------|----------|
| 0 | 19800101 | -94.822258 | 38.883761 | 1980-01-01 | 1   | 0.0 | 77.683876 | 43.286247 | 0.002620 | 8.263647 |
| 1 | 19800102 | -94.822258 | 38.883761 | 1980-01-02 | 2   | 0.0 | 78.571579 | 35.635567 | 0.002420 | 7.208553 |
| 2 | 19800103 | -94.822258 | 38.883761 | 1980-01-03 | 3   | 0.0 | 82.316978 | 39.000980 | 0.001985 | 5.930851 |

```
# Add year column. This will help us grouping analyses by year.  
df['year'] = df['timestamp'].dt.year
```

## 41.1 Find date of last frost

```
# Define variables for last frost  
lf_start_doy = 1  
lf_end_doy = 183 # Around July 15 (to be on the safe side)  
freeze_temp = 0 # Celsius  
  
# Get unique years  
unique_years = df['year'].unique()  
lf_doy = []  
  
for year in unique_years:  
    idx_year = df['year'] == year  
    idx_lf_period = (df['doy'] > lf_start_doy) & (df['doy'] <= lf_end_doy)  
    idx_frost = df['tmmn'] < freeze_temp  
    idx = idx_year & idx_lf_period & idx_frost  
  
    # Select all DOY for current year that meet all conditions.  
    # Sort in ASCENDING order. The last value was the last freezing DOY  
    all_doy_current_year = df.loc[idx, 'doy'].sort_values()  
    lf_doy.append(all_doy_current_year.iloc[-1])
```

## 41.2 Find date of first frost

```
# Define variables for first frost  
ff_start_doy = 183 # Around July 15 (to be on the safe side)  
ff_end_doy = 365  
  
# Get unique years  
ff_doy = []  
  
for year in unique_years:  
    idx_year = df['year'] == year
```

```

idx_ff_period = (df['doy'] > ff_start_doy) & (df['doy'] <= ff_end_doy)
idx_frost = df['tmmn'] < freeze_temp
idx = idx_year & idx_ff_period & idx_frost

# Select all DOY for current year that meet all conditions.
# Sort in DESCENDING order. The last value was the last freezing DOY
all_doy_current_year = df.loc[idx, 'doy'].sort_values(ascending=False)
ff_doy.append(all_doy_current_year.iloc[-1])

```

### 41.3 Find median date for first and last frost

```

# Create dataframe with the first and last frost for each year
# The easiest is to create a dictionary with the variables we already have

df_frost = pd.DataFrame({'year':unique_years,
                         'first_frost_doy':ff_doy,
                         'last_frost_doy':lf_doy})
df_frost.head(3)

```

|   | year | first_frost_doy | last_frost_doy |
|---|------|-----------------|----------------|
| 0 | 1980 | 299             | 104            |
| 1 | 1981 | 296             | 79             |
| 2 | 1982 | 294             | 100            |

```

# Print median days of the year
df_frost[['first_frost_doy','last_frost_doy']].median()

```

```

first_frost_doy      298.0
last_frost_doy       96.0
dtype: float64

```

```

# Compute median DOY and calculate date for first frost
first_frost_median_doy = df_frost['first_frost_doy'].median()
first_frost_median_date = pd.to_datetime('2000-01-01') + pd.Timedelta(first_frost_median_doy, unit='D')
print(f"Median date first frost: {first_frost_median_date.strftime('%d-%B')}")

```

```

first_frost_earliest_doy = df_frost['first_frost_doy'].min() # Min value for earliest first frost
first_frost_earliest_date = pd.to_datetime('2000-01-01') + pd.Timedelta(first_frost_earliest_doy)
print(f"Earliest date first frost on record: {first_frost_earliest_date.strftime('%d-%B')}")

# Compute median DOY and calculate date for first frost
last_frost_median_doy = df_frost['last_frost_doy'].median()
last_frost_median_date = pd.to_datetime('2000-01-01') + pd.Timedelta(last_frost_median_doy)
print(f"Median date last frost: {last_frost_median_date.strftime('%d-%B')}")

last_frost_latest_doy = df_frost['last_frost_doy'].max() # Max value for latest last frost
last_frost_latest_date = pd.to_datetime('2000-01-01') + pd.Timedelta(last_frost_latest_doy)
print(f"Latest date last frost on record: {last_frost_latest_date.strftime('%d-%B')}")

Median date first frost: 25-October
Earliest date first frost on record: 23-September
Median date last frost: 06-April
Latest date last frost on record: 30-April

```

## 41.4 Compute frost-free period

```

# Period without any risk of frost
frost_free = first_frost_earliest_doy - last_frost_latest_doy
print(f'Frost-free period: {frost_free} days')

```

Frost-free period: 146 days

## 41.5 Probability density functions

Let's first examine if a normal distribution fits the observations using probability plots, which compare the distribution of our data against the quantiles of a specified theoretical distribution (normal distribution in this case, similar to qq-plots). If the agreement is good, then this provides some support for using the selected distribution.

```

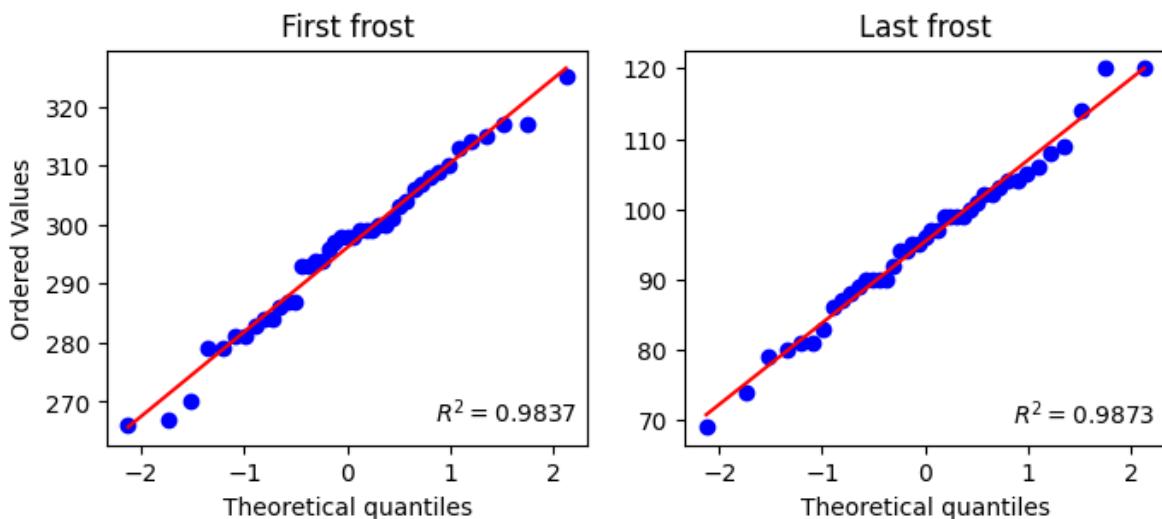
# Check distribution of data
plt.figure(figsize=(8,3))
plt.subplot(1,2,1)
stats.probplot(df_frost['first_frost_doy'], dist="norm", rvalue=True, plot=plt)

```

```

plt.title('First frost')
plt.subplot(1,2,2)
stats.probplot(df_frost['last_frost_doy'], dist="norm", rvalue=True, plot=plt)
plt.title('Last frost')
plt.ylabel('')
plt.show()

```



```

# Fit normal distributions
fitted_pdf_ff = stats.fit(stats.norm, df_frost['first_frost_doy'], bounds=((180,365),(1,25)))
print(fitted_pdf_ff.params)

fitted_pdf_lf = stats.fit(stats.norm, df_frost['last_frost_doy'], bounds=((1,180),(1,25)))
print(fitted_pdf_lf.params)

```

```

FitParams(loc=296.12195397938365, scale=13.816206118531046)
FitParams(loc=95.39034225034837, scale=11.17897667307393)

```

```

# Create vector for the normal pdf of first frost
x_ff = np.linspace(df_frost['first_frost_doy'].min(),
                    df_frost['first_frost_doy'].max(),
                    num=1000)

# Create vector for the normal pdf of last frost
x_lf = np.linspace(df_frost['last_frost_doy'].min(),

```

```

df_frost['last_frost_doy'].max(),
num=1000)

# Figure of free-frost period
plt.figure(figsize=(6,3))
plt.title('Johnson County, KS')

# Add histograms for first and last frost
plt.hist(df_frost['first_frost_doy'], bins='scott', density=True,
         label='Median first frost', facecolor=(0,0.5,1,0.25), edgecolor='navy')
plt.hist(df_frost['last_frost_doy'], bins='scott', density=True,
         label='Median last frost', facecolor=(1,0.2,0,0.25), edgecolor='tomato')

# Add median lines
plt.axvline(last_frost_median_doy, linestyle='--', color='tomato')
plt.axvline(first_frost_median_doy, linestyle='--', color='navy')

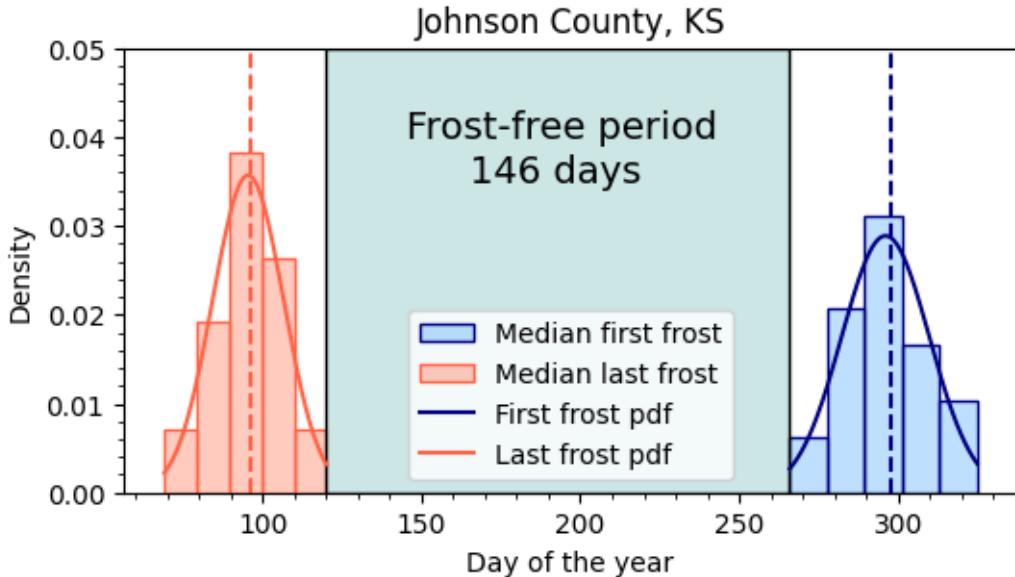
# Overlay fitted distributions to each histogram
plt.plot(x_ff, stats.norm.pdf(x_ff, *fitted_pdf_ff.params),
          color='navy', label='First frost pdf')
plt.plot(x_lf, stats.norm.pdf(x_lf, *fitted_pdf_lf.params),
          color='tomato', label='Last frost pdf')

# Add filled area to show the frost-free period
plt.fill_betweenx(np.linspace(0,0.05), last_frost_latest_doy, first_frost_earliest_doy,
                  facecolor=(0, 0.5, 0.5, 0.2), edgecolor='k')

# Add some annotations
plt.text(145, 0.04, "Frost-free period", size=14)
plt.text(165, 0.035, f"{frost_free} days", size=14)

plt.ylim([0, 0.05])
plt.xlabel('Day of the year')
plt.ylabel('Density')
plt.minorticks_on()
plt.legend()
plt.show()

```



```

# Cumulative distributions
# Create vector from 0 to x_max to plot the lognorm pdf
x = np.linspace(df_frost['first_frost_doy'].min(),
                 df_frost['first_frost_doy'].max(),
                 num=1000)

plt.figure(figsize=(10,4))
plt.suptitle('Johnson County, KS 1980-2020')

# First frost
plt.subplot(1,2,1)
plt.ecdf(df_frost['first_frost_doy'], label="Empirical CDF")
plt.plot(x_ff, stats.norm.cdf(x_ff, *fitted_pdf_ff.params), label='Theoretical CDF')
plt.title('First frost')
plt.xlabel('Day of the year')
plt.ylabel('Cumulative density')
plt.legend()

# Last frost (note the use of the complementary CDF)
plt.subplot(1,2,2)
plt.ecdf(df_frost['last_frost_doy'], complementary=True, label="Empirical CDF")
plt.plot(x_lf, 1-stats.norm.cdf(x_lf, *fitted_pdf_lf.params), label='Theoretical CDF')
plt.title('Last frost')

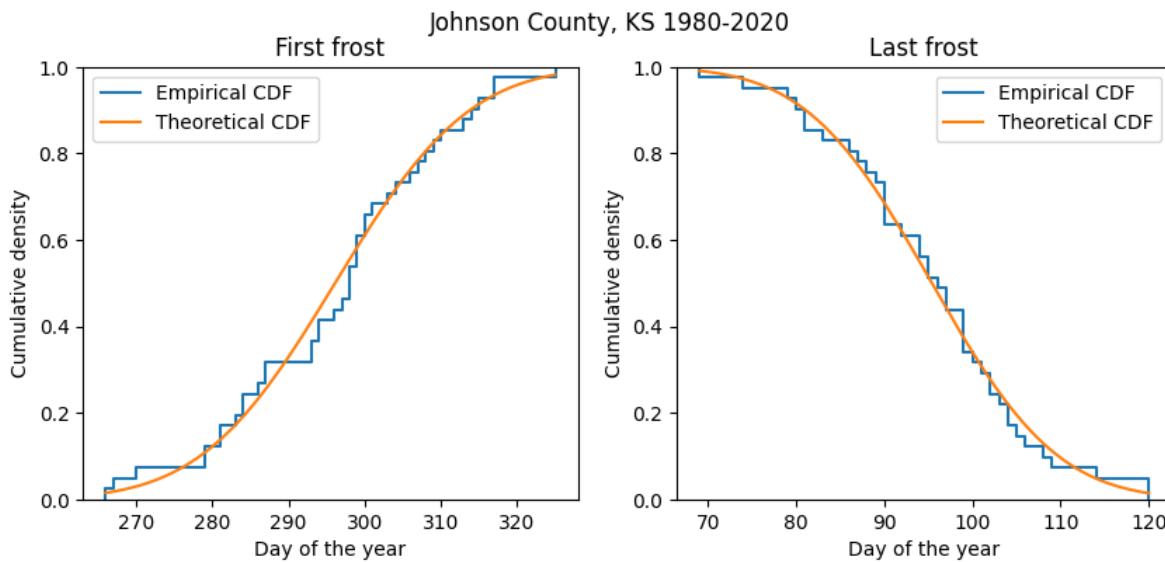
```

```

plt.xlabel('Day of the year')
plt.ylabel('Cumulative density')
plt.legend()

plt.show()

```



```

# Determine the probability of a first frost occurring on or before:
doy = 245 # September 1
stats.norm.cdf(doy, *fitted_pdf_ff.params)

# As expected, if you change the DOY for a value closer to July 1,
# the chances of frost in the north hemisphere are going to decrease to nearly zero.

```

0.00010773852588002489

```

# Determine the probability of a frost occurring on or after:
doy = 122 # May 1
stats.norm.sf(doy, *fitted_pdf_lf.params)

# As expected, if you change the DOY for a value closer to January 1,
# the chances of frost in the north hemisphere are going to increase to 1 (or 100%).

```

0.008648561213750895

**i** Note

Why did we use the complementary of the CDF for determining the probability of last frost? We used the complementary CDF (or sometimes known as the survival function, hence the syntax `.sf()`), because in most cases we are interested in knowing the probability of a frost “on or after” a given date. For instance, a farmer that is risk averse will prefer to plant corn when the chances of a late frost that can kill the entire crop overnight is very low.

## 42 Air Temperature

Seasonal changes in air temperature closely follow the sinusoidal pattern of solar radiation. So, for a given location, there is a strong relationship with the day of the year and a strong autocorrelation, which means that, typically, the air tempearature today is similar to the air temperature yesterday or tomorrow. In this exercise we will inspect the dynamics of daily minimum (TMIN) and maximum (TMAX) air temperatures. We will calculate average TMAX and TMIN for each day of the year and we will also fit a simple sinusoidal model and examine its residuals.

```
# Import modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Load sample data
df = pd.read_csv("../datasets/KS_Manhattan_6_SSW.csv")

# Keep only the columns that we need
df = df[['LST_DATE', 'T_DAILY_MIN', 'T_DAILY_AVG', 'T_DAILY_MAX']]

# Display dataset
df.head()
```

|   | LST_DATE | T_DAILY_MIN | T_DAILY_AVG | T_DAILY_MAX |
|---|----------|-------------|-------------|-------------|
| 0 | 20031001 | -9999.0     | -9999.0     | -9999.0     |
| 1 | 20031002 | 2.5         | 11.7        | 18.9        |
| 2 | 20031003 | 8.1         | 14.8        | 22.6        |
| 3 | 20031004 | 3.8         | 14.0        | 22.6        |
| 4 | 20031005 | 10.6        | 17.3        | 25.0        |

```
# Convert date string to pandas datetime format
df["LST_DATE"] = pd.to_datetime(df["LST_DATE"], format="%Y%m%d")
```

```

df["LST_DATE"].head() # Check our conversion.

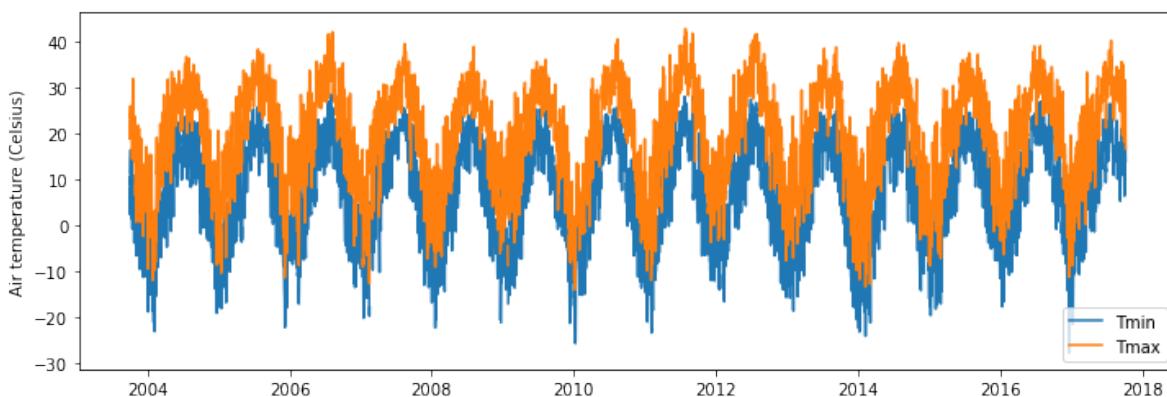
0    2003-10-01
1    2003-10-02
2    2003-10-03
3    2003-10-04
4    2003-10-05
Name: LST_DATE, dtype: datetime64[ns]

# Convert missing values represented as -9999 to NaN
df[df == -9999] = np.nan

# Add year, month, and day of the year to summarize data in future steps.
df["YEAR"] = df["LST_DATE"].dt.year
df["MONTH"] = df["LST_DATE"].dt.month
df["DOY"] = df["LST_DATE"].dt.dayofyear

# Air temperature
plt.figure(figsize=(12,4))
plt.plot(df["LST_DATE"], df["T_DAILY_MIN"], label="Tmin")
plt.plot(df["LST_DATE"], df["T_DAILY_MAX"], label="Tmax")
plt.ylabel("Air temperature (Celsius)")
plt.legend()
plt.show()

```



## 42.1 Compute annual average air temperature

To establish some predictive power in our weather generator we need to extract the deterministic and stochastic components of the signal.

```
# Get annual average Tmin and Tmax
T_avg_avg = df["T_DAILY_AVG"].median()
T_min_avg = df["T_DAILY_MIN"].median()
T_max_avg = df["T_DAILY_MAX"].median()

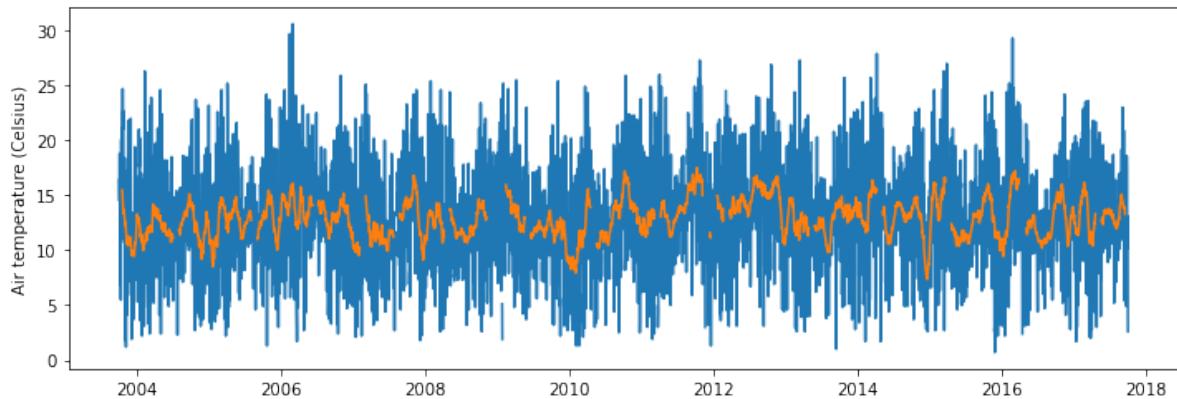
print("Tavg_avg:", T_avg_avg)
print("Tmin_avg:", T_min_avg)
print("Tmax_avg:", T_max_avg)
```

```
Tavg_avg: 14.2
Tmin_avg: 7.1
Tmax_avg: 21.3
```

## 42.2 Compute daily thermal amplitude

```
# Difference between Tmax and Tmin
T_amplitude = df["T_DAILY_MAX"] - df["T_DAILY_MIN"]

plt.figure(figsize=(12,4))
plt.plot(df["LST_DATE"], T_amplitude)
plt.plot(df["LST_DATE"], T_amplitude.rolling(window=30, center=True).mean())
plt.ylabel("Air temperature (Celsius)")
plt.show()
```



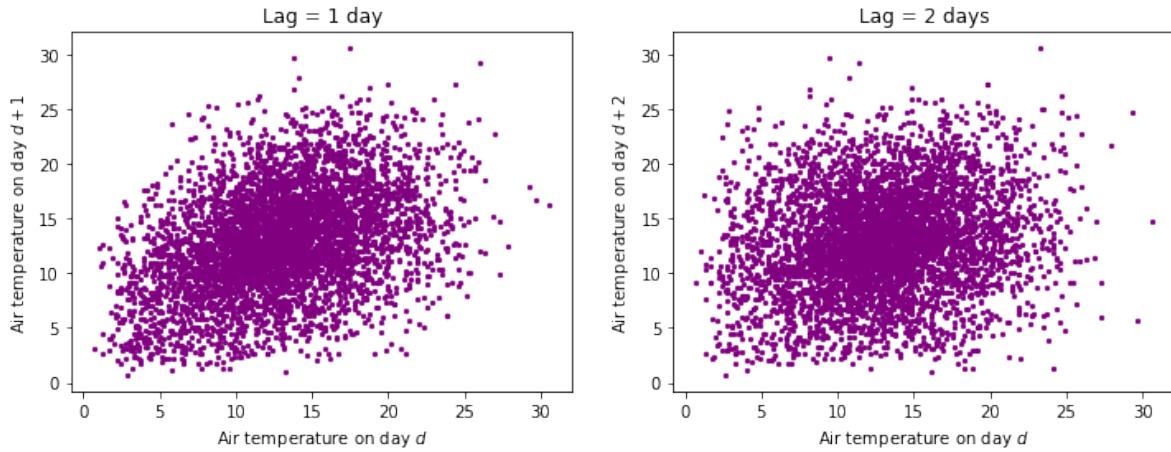
Investigate if there is autocorrelation between consecutive differences in Tmax and Tmin. If there is poor correlation, then that is a sign of randomness. If the differences are correlated, then we might need to account for this trend to better mimic the natural patterns.

```

plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.title("Lag = 1 day")
plt.scatter(T_amplitude[0:-1],T_amplitude[1:], s=5, color='purple')
plt.xlabel("Air temperature on day $d$")
plt.ylabel("Air temperature on day $d+1$")

plt.subplot(1,2,2)
plt.title("Lag = 2 days")
plt.scatter(T_amplitude[0:-2],T_amplitude[2:], s=5, color='purple')
plt.xlabel("Air temperature on day $d$")
plt.ylabel("Air temperature on day $d+2$")
plt.show()

```



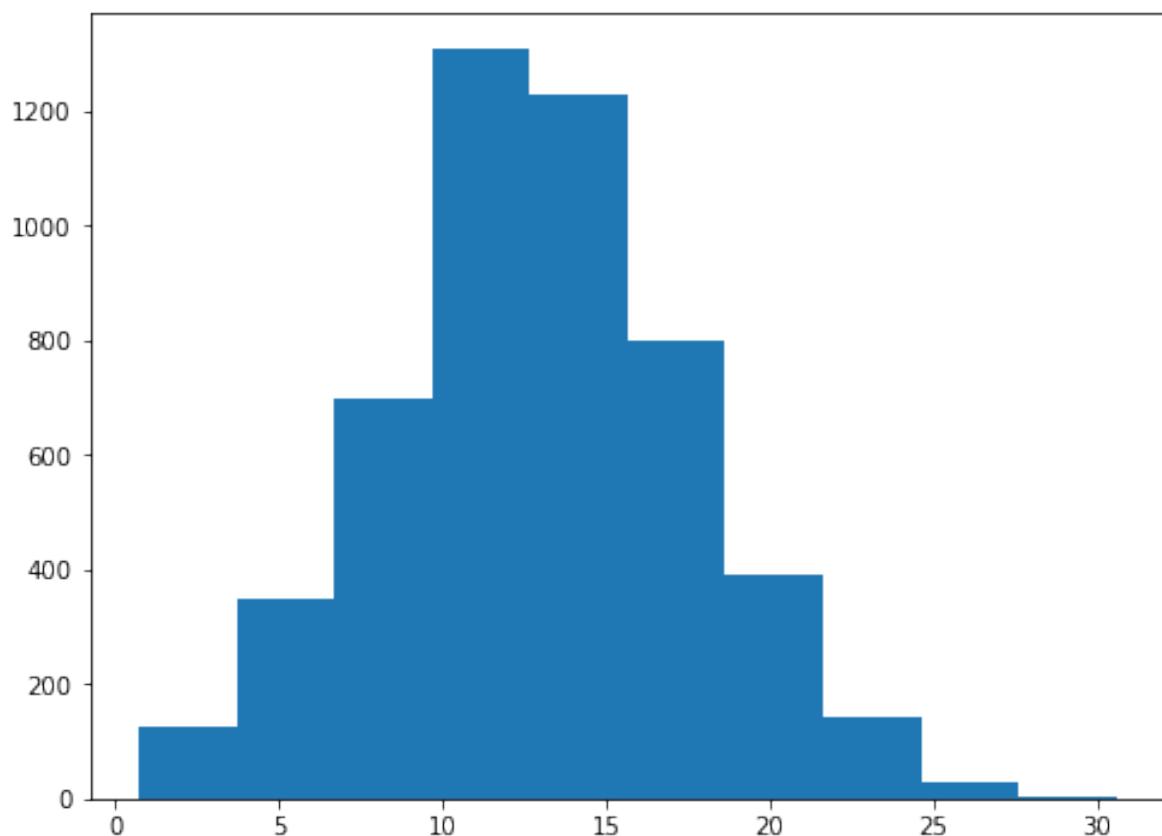
With a lag of 1 day we see a some positive trend, but it seems weak and probably not worth considering in our trivial example. This is even more clear with a lag of 2 days. This trends mean that the difference between Tmax and Tmin in one day are poorly correlated with the difference in Tmax and Tmin the next day. In principle this help us support our choice of modelling these fluctuations using a random process.

Note that the differences may be related to another variable, such as rainfall and solar radiation.

Let's check out the histogram so that we can have a sound idea of the type of random process that we need to simulate.

```
# Histogram
plt.figure(figsize=(8,6))
plt.title("Differences between Tmax and Tmin")
plt.hist(T_amplitude)
plt.show()
```

Differences between Tmax and Tmin



```
# Compute normal distribution parameters
T_amplitude_mean = T_amplitude.mean()
T_amplitude_std = T_amplitude.std()

print(round(T_amplitude_mean,1), 'Celsius')
print(round(T_amplitude_std,1), 'Celsius')
```

12.9 Celsius  
4.7 Celsius

## 42.3 Examine residuals

```
mean_tmax = df["T_DAILY_MAX"].rolling(window=30, center=True).mean()
mean_tmin = df["T_DAILY_MIN"].rolling(window=30, center=True).mean()

residuals_tmax = mean_tmax - df["T_DAILY_MAX"]
residuals_tmin = mean_tmin - df["T_DAILY_MIN"]

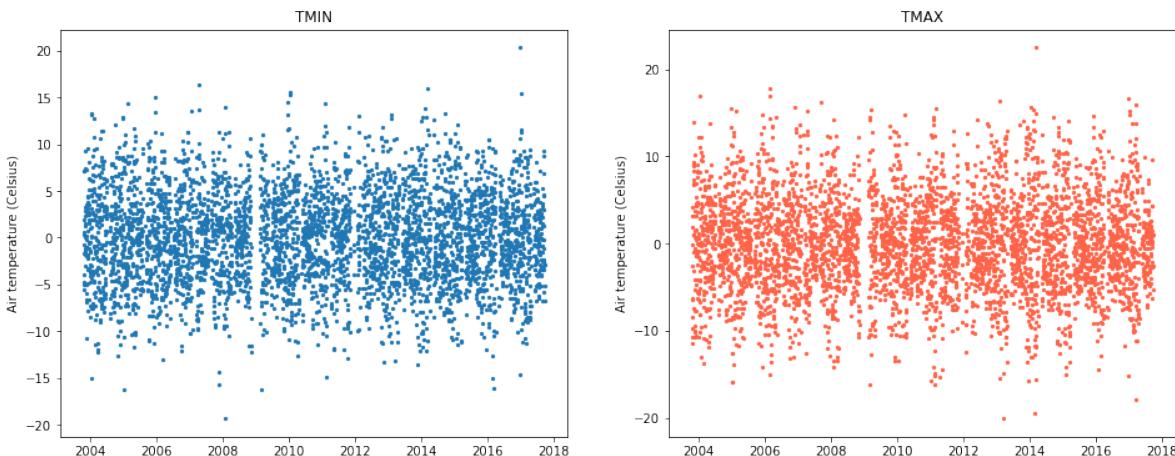
print("Residuals tmax: Mean=",residuals_tmax.mean(), "Std:",residuals_tmax.std())
print("Residuals tmin: Mean=",residuals_tmin.mean(), "Std:",residuals_tmin.std())

plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.scatter(df["LST_DATE"],residuals_tmin, s=5)
plt.title('TMIN')
plt.ylabel("Air temperature (Celsius)")

plt.subplot(1,2,2)
plt.scatter(df["LST_DATE"],residuals_tmax, s=5, color='tomato')
plt.title('TMAX')
plt.ylabel("Air temperature (Celsius)")

plt.show()
```

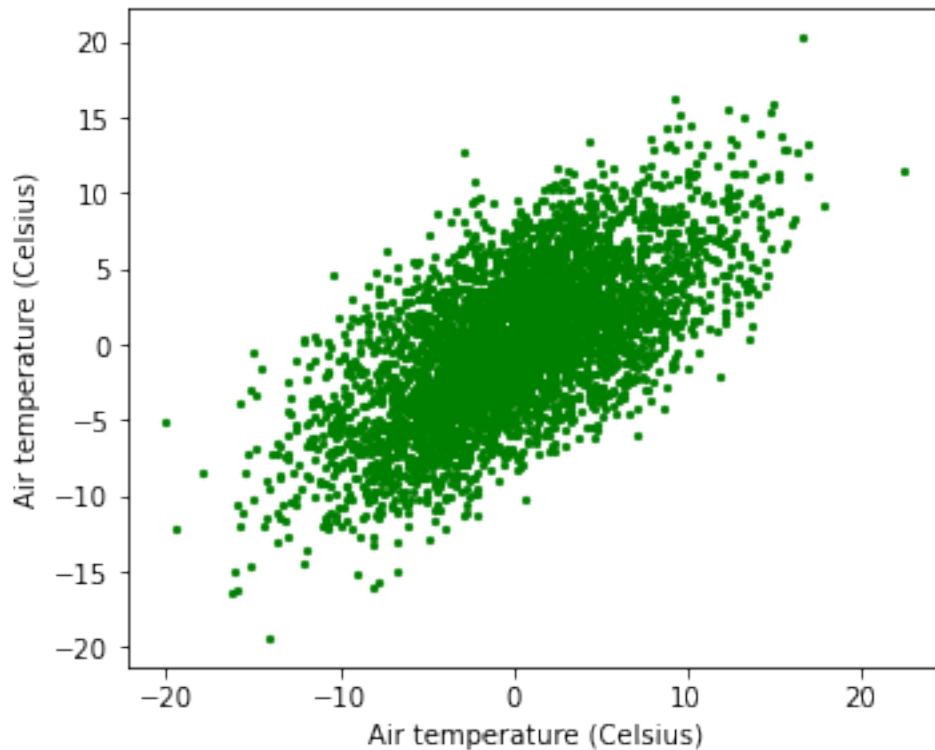
```
Residuals tmax: Mean= -0.019181784247878537 Std: 5.414114625572652
Residuals tmin: Mean= -0.0027578743096251766 Std: 4.879320798713006
```



```

# Investigate correlation of residuals of TMIN and TMAX
plt.figure(figsize=(12,10))
plt.subplot(2,2,1)
plt.scatter(residuals_tmax, residuals_tmin, s=5, color='green')
plt.xlabel("Air temperature (Celsius)")
plt.ylabel("Air temperature (Celsius)")
plt.show()

```



## 42.4 Compute Tmax and Tmin for each DOY

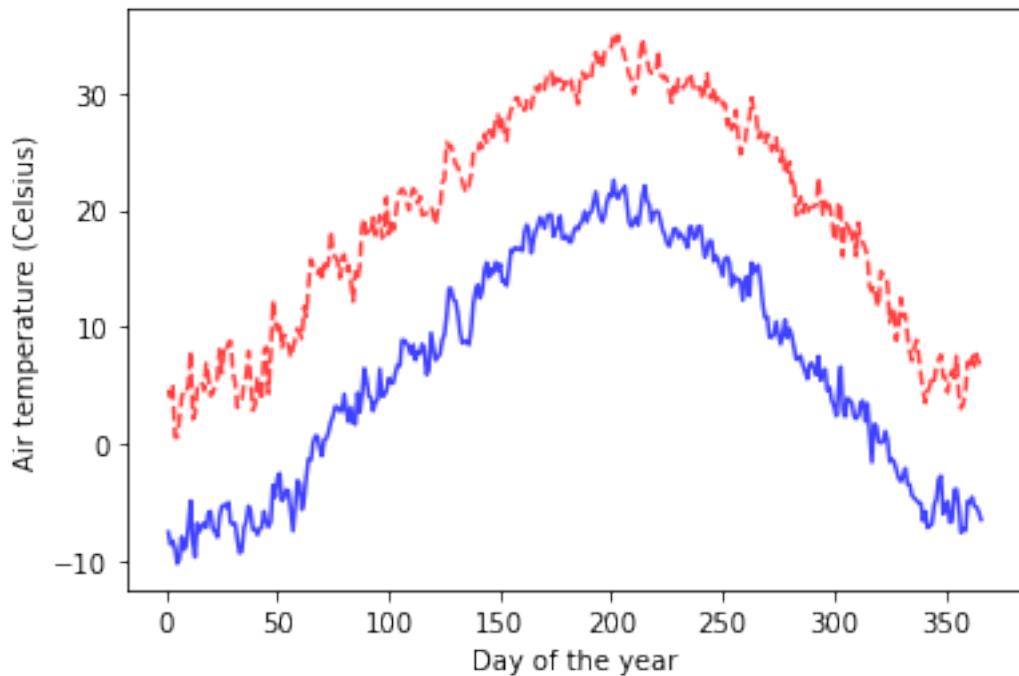
```

# Generate smooth
df_grouped = df.groupby(["DOY"]).mean()

plt.plot(df_grouped.index, df_grouped["T_DAILY_MIN"], '-b', alpha=0.75, label='TMIN')
plt.plot(df_grouped.index, df_grouped["T_DAILY_MAX"], '--r', alpha=0.75, label='TMAX')
plt.xlabel('Day of the year')
plt.ylabel("Air temperature (Celsius)")

```

```
plt.show()
```



```
# Occurrence of maximum values (on average)
print(df_grouped["T_DAILY_MIN"].rolling(window=15, center=True).mean().idxmax())
print(df_grouped["T_DAILY_MAX"].rolling(window=15, center=True).mean().idxmax())
```

199

199

```
# Occurrence of minimum values (on average)
print(df_grouped["T_DAILY_MIN"].rolling(window=15, center=True).mean().idxmin())
print(df_grouped["T_DAILY_MAX"].rolling(window=15, center=True).mean().idxmin())
```

8

8

## 42.5 Compute annual thermal amplitude

To compute the amplitude we will first smooth the air temperature signal using a centered moving average. Then we have two options, computing the amplitude from records of `Here` there are two options: 1) computing the amplitude using a smoothed version of `Tmin` and `Tmax`, or 2) simply computing the average amplitude of each day on record. Let's try both and see if there is any difference. We should favor the easiest approach.

For the first approach, we will estimate the annual average thermal amplitude by first computing the amplitude for each day of the year, and then calculate the annual average. This approach removes some noise before calculating the amplitude.

```
# Compute Tmax average amplitude by using de-noised signal
T_max_max = df_grouped["T_DAILY_MAX"].rolling(window=15, center=True).mean().max()
T_max_min = df_grouped["T_DAILY_MAX"].rolling(window=15, center=True).mean().min()
T_max_amplitude = ((T_max_max - T_max_min)/2)
print(T_max_amplitude)
```

14.834194139194146

```
# Compute Tmin average amplitude by using de-noised signal
T_min_max = df_grouped["T_DAILY_MIN"].rolling(window=15, center=True).mean().max()
T_min_min = df_grouped["T_DAILY_MIN"].rolling(window=15, center=True).mean().min()
T_min_amplitude = ((T_min_max - T_min_min)/2)
print(T_min_amplitude)
```

14.545347985347988

The amplitude of `Tmax` and `Tmin` are remarkably similar, which means that we can probably take advantage of this fact to model air temperature using the same amplitude parameter.

## 42.6 Build model for air temperature

A sinusoidal model is probably adequate to mimic the seasonal dynamics. This is the deterministic part of our model. The difference between our model and observations we will be attributed to random noise, or to more complicated relationships that go beyond the scope of this exercise.

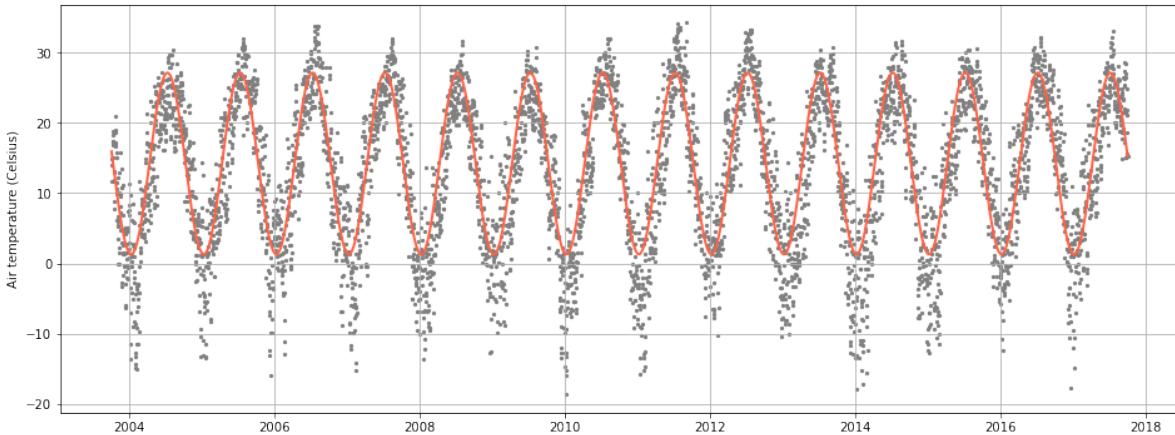
$$T(t) = T_{avg} + A \sin \left[ 2\pi \left( \frac{DOY}{365} + \phi + \frac{\pi}{2} \right) \right] + \epsilon$$

```
# Get amplitude of the sinusoidal pattern
dates = df["LST_DATE"]

# Get day of the year for each input date
doy = df["LST_DATE"].dt.dayofyear

# Set model parameters based on our previous findings
T_avg = 14.2
A = 12.9
phase = 8
T_pred = T_avg + A * np.sin(2*np.pi*(doy-phase)/365) - np.pi/2

plt.figure(figsize=(16,6))
plt.scatter(df["LST_DATE"], df["T_DAILY_AVG"], s=5, color='gray', label="Observed TAVG")
plt.plot(dates, T_pred, label="Predicted TAVG", color='tomato', linewidth=2)
plt.ylabel("Air temperature (Celsius)")
plt.grid()
plt.show()
```



```
# Compute residuals between observed and predicted
TAVG_residuals = df["T_DAILY_AVG"] - T_pred
plt.figure(figsize=(16,6))
plt.scatter(doy, TAVG_residuals, s=5, color='purple')
```

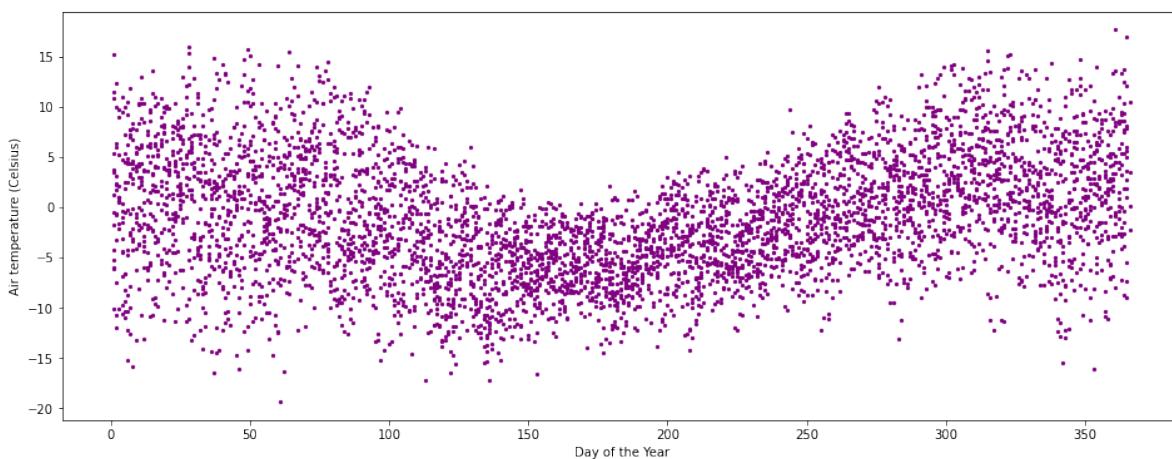
```

plt.xlabel('Day of the Year')
plt.ylabel("Air temperature (Celsius)")
plt.show()

# Compute mean absolute error
MAE = np.mean(np.abs(TAVG_residuals))

# Object-oriented computation
#MAE = TAVG_residuals.abs().mean()
print('Mean absolute error:', round(MAE,1) , 'Celsius')

```



Mean absolute error: 4.9 Celsius

Our model does a reasonable job predicting air temperature with an error of 4 degrees Celsius. Depending on the application the magnitude of the error may be acceptable or unacceptable. Clearly our model struggles to compute accurate daily average temperature values during the winter, spring, and fall seasons and is also not capturing the year-to-year variability. The model parameters were not optimized in this case, but rather estimated from local observations.

## 42.7 Practice

- What mean absolute error do you obtain if the amplitude is increased by 3 degrees Celsius in an attempt to better capture low temperatures?
- What modifications would you implement to reduce the error during winter, fall, and spring seasons?

## 43 Potential Evapotranspiration

Soil evaporation and plant transpiration are two important components of the soil water balance. Evapotranspiration is the sum of these two evaporative losses (one from the soil and the other from the plant stomates). Measuring and modeling soil evaporation and transpiration individually is usually harder than estimating the combined total of these two variables. For instance, if we measure the soil water content in the soil profile yesterday and today, assuming that there was not rainfall and that deep drainage was negligible, we could assume that the change in soil water storage was caused by evapotrasnpiration. In other words, the water loss from the soil was (mainly) by evaporation and plant transpiration.

If we don't need to know the exact partition between evaporation and trasnpiration, then this simple method will provide an estimation of the evapotranspiration between yesterday and today. This would be the **actual evapotranspiration**.

In agriculture, there is another term, the **potential evapotrasnpiration** (PET), which refers to the potential evaporative losses, typically by an reference crop of idealized conditions (e.g. green grass of 8–15 cm height) actively growing under non limiting soil moisture conditions and under the influence of the local weather. This value is used as reference of the maximum water loss from the system due to soil evaporation and plant transpiration.

Knowing the potential evaporation is useful to characterize climate regimes, usually the ratio of annual precipitation to annual PET. Another common use of PET, is in combination with empirical coefficients that can be used to weigh the PET of the reference crop to another crop.

In this notebook we will use daily weather observations to explore several models. Some models, like the Thornthwaite model, work at a monthly scale, but because daily meteorological data is ubiquitous we will only focus on daily computations. Still monthly approaches can be valuable in locations where there are limited or none weather stations.

Over the year scientists have developed multiple ways of estimating PET. A rather convenie

Potetnial evapotrasnpiration is measured using field lysimeters and modeled using meterological information. One of the first models was proposed by John Dalton in 1802, and the most widely used model was proposed by Howard Penman with modifications by John Monteith in the 1950s. In 1998, the Food and Agriculture Organization developed a standard method for computing PET using the Penman-Monteith method.

```
# Import modules
import pandas as pd
import numpy as np
from bokeh.plotting import figure,show,output_notebook
output_notebook()
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_1

Because potential evapotranspiration represents potential evaporative losses, the atmospheric vapor pressure deficit is a common term in many models. The vapor pressure deficit is computed by subtracting the actual vapor pressure from the saturation vapor pressure at the same ambient temperature. Vapor pressure deficit can be easily approximated using air temperature and relative humidity using Tetens equation for the saturation vapor pressure.

In order to use the Tetens equation in multiple functions, we will first define the Tetens function separately. You can include the Tetens equation as part of the PET function, in order to create a single, stand-alone function for your projects. here this approach is more convenient since we are comparing multiple models.

```
# Auxiliary functions
tetens = lambda T: 0.6108*np.exp(17.27*T/(T+237.3))

# Plot tetens function to illustrate the dependence of saturation vapor to temperature
air_temperature_range = np.arange(50)
f = figure(width=400, height=300)
f.line(air_temperature_range, tetens(air_temperature_range))
f.xaxis.axis_label = 'Air Temperature (Celsius)'
f.yaxis.axis_label = 'Saturation Vapor Pressure (kPa)'
show(f)
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_1

## 43.1 Dalton model

In 1802 John Dalton proposed a simple model for predicting potential evaporation based on wind speed and vapor pressure deficit. This equation works well on open water bodies, but it does not account for the effect of plants and the presence of soils. Several equations have been based on the formulation proposed by Dalton, in which additional coefficients and exponents aim at better predicting local conditions.

Dalton started as a meteorologist, frequently recording atmospheric conditions in his own journals. The equation proposed by Dalton is considered a mass transfer model, the name emphasizing that water vapor moves along a vapor gradient and that this gradient can be maintained or enhanced by wind replacing the saturated air near the evaporating surface with new air volume containing lower water vapor.

$$E = u(e_s - e_a)$$

$u$  is the wind speed in m/s

$e_s$  is the atmospheric saturation vapor pressure in kPa

$e_a$  is the actual atmospheric vapor pressure

```
## John Dalton (1802)

def dalton(T_min,T_max,RH_min,RH_max,wind_speed):
    e_sat = (tetens(T_min) + tetens(T_max)) / 2
    e_atm = (tetens(T_min)*(RH_max/100) + tetens(T_max)*(RH_min/100)) / 2
    PE = (3.648 + 0.7223*wind_speed)*(e_sat - e_atm)
    return PE
```

## 43.2 Romanenko model

```
## Romanenko (1961)

def romanenko(T_min,T_max,RH_min,RH_max):
    T_avg = (T_min + T_max)/2
    RH_avg = (RH_min + RH_max)/2
    PET = 0.00006*(25 + T_avg)**2*(100 - RH_avg)
    return PET
```

### 43.3 Penman model

```
## Penman (1948)

def penman(T_min,T_max,RH_min,RH_max,wind_speed):
    e_sat = (tetens(T_min) + tetens(T_max)) / 2
    e_atm = (tetens(T_min)*(RH_max/100) + tetens(T_max)*(RH_min/100)) / 2
    PET = (2.625 + 0.000479/wind_speed)*(e_sat - e_atm)
    return PET
```

### 43.4 Jensen model

```
## Jensen-Haise (1963)

def jensen_haise(T_min,T_max,solar_rad):
    T_avg = (T_min + T_max)/2
    PET = 0.0102*(T_avg+3)*solar_rad
    return PET
```

### 43.5 Hargreaves model

$$PET = 0.0023 R_a (T_{avg} + 17.8) \sqrt{(T_{max} - T_{min})}$$

$R_a$  is the extraterrestrial solar radiation ( $MJ/m^2$ )

$T_{max}$  is the maximum daily air temperature

$T_{min}$  is the minimum daily air temperature

$T_{avg}$  is the average daily air temperature

```
## Hargreaves (1982)

def hargreaves(T_min,T_max,doy,latitude):

    # Computation of extra-terrestrial solar radiation
    dr = 1 + 0.033 * np.cos(2 * np.pi * doy/365) # Inverse relative distance Earth-Sun
    phi = np.pi / 180 * latitude # Latitude in radians
    d = 0.409 * np.sin((2 * np.pi * doy/365) - 1.39) # Solar declination
```

```

omega = np.arccos(-np.tan(phi) * np.tan(d)) # Sunset hour angle
Gsc = 0.0820 # Solar constant
Ra = 24 * 60 / np.pi * Gsc * dr * (omega * np.sin(phi) * np.sin(d) + np.cos(phi) * np.
T_avg = (T_min + T_max)/2
PET = 0.0023 * Ra * (T_avg + 17.8) * (T_max - T_min)**0.5
return PET

```

### 43.5.1 Priestley-Taylor model

This model was developed with the intention of reducing the number of variables required to compute the potential evapotranspiration in regions with limited environmental observations as an alternative to the model proposed by Penman-Monteith. Thus, several terms of the PM equation are condensed into an empirical constant. The psychrometric constant is computed as:

$$\gamma = \frac{C_p P}{\epsilon \lambda} \approx 6.6510^{-4} P$$

$\gamma$  = is the psychrometric constant in kPa/°C

$C_p$  is the specific heat at constant pressure 0.001013 (MJ kg<sup>-1</sup> °C<sup>-1</sup>)

$\epsilon$  is the ratio molecular weight of water vapour/dry air = 0.622.

$\lambda$  is the latent heat of vaporization, 2.45 MJ kg<sup>-1</sup>

$P$  is the atmospheric pressure in kPa either computed from the station altitude or measured by a barometer.

The term  $(R_n - G)$  represents the available energy

```

## Priestley-Taylor (1972)

def priestley_taylor(T_min,T_max,solar_rad,altitude):
    T_avg = (T_min + T_max)/2
    atm_pressure = 101.3 * ((293 - 0.0065 * altitude)/293)**5.26 # kPa
    gamma = 0.000665 * atm_pressure # Psychrometric constant Cp/(2.45 * 0.622) = 0.000665
    delta = 4098 * (0.6108 * np.exp(17.27 * T_avg / (T_avg + 237.3))) / (T_avg + 237.3)*
    soil_heat_flux = 0;
    alpha = 0.5
    PET = alpha*delta/(delta+gamma)*(solar_rad-soil_heat_flux)
    return PET

```

## 43.6 Penman-Monteith model

```
def penman_monteith(T_min,T_max,RH_min,RH_max,solar_rad,wind_speed,doy,latitude,altitude):
    T_avg = (T_min + T_max)/2
    atm_pressure = 101.3 * ((293 - 0.0065 * altitude) / 293)**5.26 # Can be also obtained
    Cp = 0.001013; # Approx. 0.001013 for average atmospheric conditions
    epsilon = 0.622
    Lambda = 2.45
    gamma = (Cp * atm_pressure) / (epsilon * Lambda) # Approx. 0.000665

    ##### Wind speed
    wind_height = 1.5 # Most common height in meters
    wind_speed_2m = wind_speed * (4.87 / np.log((67.8 * wind_height) - 5.42)) # Eq. 47, FAO-56

    ##### Air humidity and vapor pressure
    delta = 4098 * (0.6108 * np.exp(17.27 * T_avg / (T_avg + 237.3))) / (T_avg + 237.3)*
    e_temp_max = 0.6108 * np.exp(17.27 * T_max / (T_max + 237.3)) # Eq. 11, //FAO-56
    e_temp_min = 0.6108 * np.exp(17.27 * T_min / (T_min + 237.3))
    e_saturation = (e_temp_max + e_temp_min) / 2
    e_actual = (e_temp_min * (RH_max / 100) + e_temp_max * (RH_min / 100)) / 2

    ##### Solar radiation
    dr = 1 + 0.033 * np.cos(2 * np.pi * doy/365) # Eq. 23, FAO-56
    phi = np.pi / 180 * latitude # Eq. 22, FAO-56
    d = 0.409 * np.sin((2 * np.pi * doy/365) - 1.39)
    omega = np.arccos(-np.tan(phi) * np.tan(d))
    Gsc = 0.0820 # Approx. 0.0820
    Ra = 24 * 60 / np.pi * Gsc * dr * (omega * np.sin(phi) * np.sin(d) + np.cos(phi) * np.cos(phi))

    # Clear Sky Radiation: Rso (MJ/m2/day)
    Rso = (0.75 + (2 * 10**-5) * altitude) * Ra # Eq. 37, FAO-56

    # Rs/Rso = relative shortwave radiation (limited to <= 1.0)
    alpha = 0.23 # 0.23 for hypothetical grass reference crop
    Rns = (1 - alpha) * solar_rad # Eq. 38, FAO-56
    sigma = 4.903 * 10**-9
    maxTempK = T_max + 273.16
    minTempK = T_min + 273.16
    Rnl = sigma * (maxTempK**4 + minTempK**4) / 2 * (0.34 - 0.14 * np.sqrt(e_actual)) *
    Rn = Rns - Rnl # Eq. 40, FAO-56
```

```

# Soil heat flux density
soil_heat_flux = 0 # Eq. 42, FAO-56 G = 0 for daily time steps [MJ/m2/day]

# ETo calculation
PET = (0.408 * delta * (solar_rad - soil_heat_flux) + gamma * (900 / (T_avg + 273)) *
return np.round(PET,2)

```

## 43.7 Data

In this section we will use real data to test the different PET models.

```

# Import data
df = pd.read_csv('../datasets/acme_ok_daily.csv')
df['Date'] = pd.to_datetime(df['Date'], format='%m/%d/%y %H:%M')
df.head()

```

|   | Date       | DOY | TMAX      | TMIN      | RAIN  | HMAX | HMIN  | ATOT | W2AVG    | ETgrass  |
|---|------------|-----|-----------|-----------|-------|------|-------|------|----------|----------|
| 0 | 2005-01-01 | 1   | 21.161111 | 14.272222 | 0.00  | 97.5 | 65.97 | 4.09 | 5.194592 | 1.976940 |
| 1 | 2005-01-02 | 2   | 21.261111 | 4.794444  | 0.00  | 99.3 | 77.37 | 4.11 | 3.428788 | 1.302427 |
| 2 | 2005-01-03 | 3   | 5.855556  | 3.477778  | 2.54  | 99.8 | 98.20 | 2.98 | 3.249973 | 0.349413 |
| 3 | 2005-01-04 | 4   | 4.644444  | 0.883333  | 7.62  | 99.6 | 98.50 | 1.21 | 3.527137 | 0.288802 |
| 4 | 2005-01-05 | 5   | 0.827778  | -9.172222 | 24.13 | 99.4 | 86.80 | 1.65 | NaN      | 0.367956 |

```

dalton_pred = dalton(df['TMIN'], df['TMAX'], df['HMIN'], df['HMAX'], df['W2AVG'])
penman_pred = penman(df['TMIN'], df['TMAX'], df['HMIN'], df['HMAX'], df['W2AVG'])

# Plot models
f = figure(width=600, height=400, x_axis_type="datetime")
f.line(df['Date'], dalton_pred, legend_label='Dalton', line_color='tomato')
f.line(df['Date'], penman_pred, legend_label='Penman')
show(f)

```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_e

## 43.8 Practice

- Compute the potential evaporation or potential evapotranspiration with the other models. Which models do you think produce the most reasonable results for ET predictions over terrestrial ecosystems?
- Using the Penman-Monteith model, what is the impact of wind speed? By how many millimeters per day would the potential ET increased when incrementing the wind speed by 1 m/s?

## 43.9 References

Dalton J (1802) Experimental essays on the constitution of mixed gases; on the force of steam of vapour from waters and other liquids in different temperatures, both in a Torricellian vacuum and in air on evaporation and on the expansion of gases by heat. Mem Manch Lit Philos Soc 5:535–602

Hargreaves G (1989) Preciseness of estimated potential evapotranspiration. J Irrig Drain Eng 115(6):1000–1007

Penman HC (1948) Natural evaporation from open water, bare soil and grass. Proc R Soc Lond Ser A 193:120–145

Thornthwaite, C.W., 1948. An approach toward a rational classification of climate. Geographical review, 38(1), pp.55-94.

McMahon, T.A., Finlayson, B.L. and Peel, M.C., 2016. Historical developments of models for estimating evaporation using standard meteorological data. Wiley Interdisciplinary Reviews: Water, 3(6), pp.788-818.

## 44 Growing degree days

Growing Degree Days (GDD) is a key and widely used concept in agriculture for predicting plant growth stages. In simple terms, GDD measures heat accumulation by calculating the average temperature of a day minus a base temperature, typically the lowest temperature at which a specific crop can grow. This metric is based on the principle that plant growth and development mostly occurs within a specific temperature range. In more advanced computations, researchers typically adopt a set of minimum, optimal, and maximum temperatures for different crop stages. The concept of GDD helps farmers, agronomists, and researchers better predict growth stages, like flowering, by using a “thermal time” instead of relying on calendar days.

For instance, in corn, a base temperature of  $10^{\circ}\text{C}$  is often used, so if the average daily temperature is  $15^{\circ}\text{C}$ , then 5 GDDs are accumulated for that day. Thermal time has units of C-d (or Cd).

In this exercise we will implement non-vectorized and vectorized versions of two common methods to compute GDD and then we will use it with a real weather dataset.

**Method 1** only considers  $T_{base}$ . This the most widely used method. It assumes a linear and unrestricted accumulation of heat units when  $T_{avg} > T_{base}$

**Method 2** considers  $T_{base}$ ,  $T_{opt}$ , and  $T_{upper}$ . This method is more realistic, but it requires specifying three cardinal temperatures. Maximum accumulation of heat units is when  $T_{avg} = T_{opt}$ . When  $T_{avg}$  is below or above  $T_{opt}$ , the number of heat units accumulated is adjusted linearly.

**Vectorization** refers to the practice of applying operations to entire arrays or sequences of data at once, rather than using explicit loops. This is achieved through NumPy’s powerful array objects and functions, which are designed to operate on arrays efficiently. When a function is vectorized, it can perform element-wise operations on arrays, leading to significantly faster execution and cleaner code.

```
# Import modules
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
```

```

# Define cardinal temperatures for corn
T_base = 10
T_opt = 28
T_upper = 38

# Load some weather data
df = pd.read_csv('../datasets/gypsum_2018_daily.csv', parse_dates=['TIMESTAMP'])

# Display a few rows to inspect dataframe
df.head()

```

|   | TIMESTAMP  | STATION | PRESSUREAVG | PRESSUREMAX | PRESSUREMIN | SLPAVG | TEMP   |
|---|------------|---------|-------------|-------------|-------------|--------|--------|
| 0 | 2018-01-01 | Gypsum  | 99.44       | 100.03      | 98.73       | 104.44 | -15.15 |
| 1 | 2018-01-02 | Gypsum  | 99.79       | 100.14      | 99.40       | 104.88 | -16.48 |
| 2 | 2018-01-03 | Gypsum  | 98.87       | 99.52       | 97.94       | 103.81 | -11.03 |
| 3 | 2018-01-04 | Gypsum  | 98.22       | 98.54       | 97.90       | 102.99 | -5.83  |
| 4 | 2018-01-05 | Gypsum  | 98.10       | 98.42       | 97.75       | 102.88 | -4.73  |

```

# Find number of missing values
df['TEMP2MAVG'].isna().sum()

```

1

```

# Replace missing values using linear method
df['TEMP2MAVG'].interpolate(method='linear', inplace=True)

# Check that missing values were replaced
df['TEMP2MAVG'].isna().sum()

```

0

```

# Select data for growing season
planting_date = pd.to_datetime('2018-04-01')
harvest_date = pd.to_datetime("2018-08-15")

```

```

# Select growing season weather
idx_season = (df['TIMESTAMP'] >= planting_date) & (df['TIMESTAMP'] <= harvest_date)

```

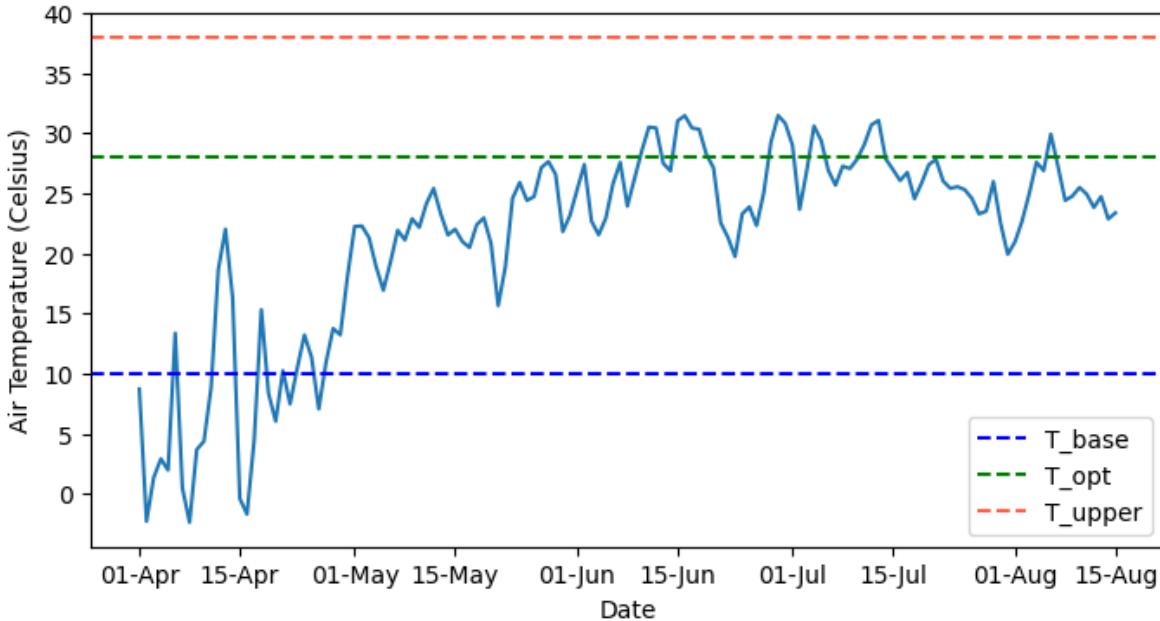
```
df_season = df[idx_season].reset_index(drop=True)
print('Growing season length:', df_season.shape[0], 'days')
```

Growing season length: 137 days

```
# Display Dataframe for the growing season
df_season.head(3)
```

|   | TIMESTAMP  | STATION | PRESSUREAVG | PRESSUREMAX | PRESSUREMIN | SLPAVG | TEMP  |
|---|------------|---------|-------------|-------------|-------------|--------|-------|
| 0 | 2018-04-01 | Gypsum  | 97.48       | 98.18       | 96.76       | 101.93 | 8.70  |
| 1 | 2018-04-02 | Gypsum  | 97.94       | 98.23       | 97.46       | 102.62 | -2.30 |
| 2 | 2018-04-03 | Gypsum  | 96.42       | 97.55       | 95.47       | 101.01 | 1.34  |

```
# Inspect temperature data
plt.figure(figsize=(8,4))
plt.plot(df_season['TIMESTAMP'], df_season['TEMP2MAVG'])
plt.axhline(T_base, linestyle='--', color='blue', label='T_base')
plt.axhline(T_opt, linestyle='--', color='green', label='T_opt')
plt.axhline(T_upper, linestyle='--', color='tomato', label='T_upper')
plt.xlabel('Date')
plt.ylabel('Air Temperature (Celsius)')
plt.legend()
fmt = mdates.DateFormatter('%d-%b')
plt.gca().xaxis.set_major_formatter(fmt)
plt.show()
```



#### 44.1 Example using non-vectorized functions

The following functions can only accept one value of  $T_{avg}$  at the time. Which means that to compute multiple values in an array we would need to implement a loop. The advantage of this method is its simplicity and clarity.

```
# Method 1: T_base only
def gdd_method_1(T_avg,T_base,dt=1):
    if T_avg < T_base:
        GDD = 0
    else:
        GDD = (T_avg - T_base)*dt

    return GDD

# Method 2: T_base, T_opt, and T_upper (Linear)
def gdd_method_2(T_avg,T_base,T_opt,T_upper,dt=1):
    if T_avg <= T_base:
        GDD = 0

    elif T_base < T_avg < T_opt:
        GDD = (T_avg - T_base)*dt
    else:
        GDD = (T_upper - T_avg)*dt

    return GDD
```

```

        GDD = (T_avg - T_base)*dt

    elif T_opt <= T_avg < T_upper:
        GDD = (T_upper - T_avg)/(T_upper - T_opt)*(T_opt - T_base)*dt

    else:
        GDD = 0

    return GDD

# Test that functions are working as expected
T_avg = 25

print(gdd_method_1(T_avg, T_base))
print(gdd_method_2(T_avg, T_base, T_opt, T_upper))

15
15

# Compute growing degree days

# Create empty arrays to append function values
GDD_1 = []
GDD_2 = []

# Iterate over each row
for k, row in df_season.iterrows():
    GDD_1.append(gdd_method_1(row['TEMP2MAVG'], T_base))
    GDD_2.append(gdd_method_2(row['TEMP2MAVG'], T_base, T_opt, T_upper))

# Add arrays as new dataframe columns
df_season['GDD_1'] = GDD_1
df_season['GDD_2'] = GDD_2

df_season['GDD_1_cum'] = df_season['GDD_1'].cumsum()
df_season['GDD_2_cum'] = df_season['GDD_2'].cumsum()

# Display resulting dataframe (new columns are at the end)
df_season.head(3)

```

|   | TIMESTAMP  | STATION | PRESSUREAVG | PRESSUREMAX | PRESSUREMIN | SLPAVG | TEMP  |
|---|------------|---------|-------------|-------------|-------------|--------|-------|
| 0 | 2018-04-01 | Gypsum  | 97.48       | 98.18       | 96.76       | 101.93 | 8.70  |
| 1 | 2018-04-02 | Gypsum  | 97.94       | 98.23       | 97.46       | 102.62 | -2.30 |
| 2 | 2018-04-03 | Gypsum  | 96.42       | 97.55       | 95.47       | 101.01 | 1.34  |

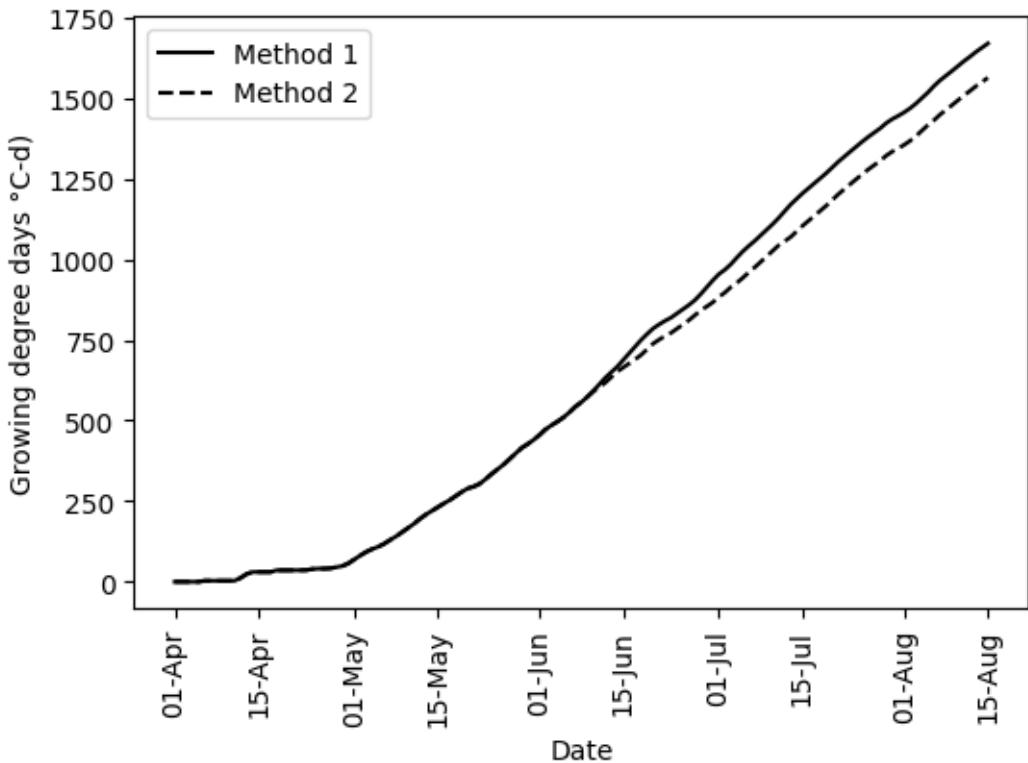
```
diff_methods = np.round(df_season['GDD_1_cum'].iloc[-1] - df_season['GDD_2_cum'].iloc[-1])
print('The difference between methods is',diff_methods, 'degree-days')
```

The difference between methods is 106.0 degree-days

```
# Create figure
plt.figure(figsize=(6,4))

plt.plot(df_season['TIMESTAMP'], df_season['GDD_1'].cumsum(), '-k', label='Method 1')
plt.plot(df_season['TIMESTAMP'], df_season['GDD_2'].cumsum(), '--k', label='Method 2')
plt.xlabel('Date')
plt.xticks(rotation=90)
# plt.ylabel(u'Growing degree days (\N{DEGREE SIGN}C-d)')
plt.ylabel(f'Growing degree days {chr(176)}C-d')
plt.legend()
fmt = mdates.DateFormatter('%d-%b')
plt.gca().xaxis.set_major_formatter(fmt)

plt.show()
```



## 44.2 Example using vectorized functions

```

def gdd_method_1_vect(T_avg, T_base, dt=1):
    """Vectorized function for computing GDD using method 1"""

    # Pre-allocate the GDD array with NaNs
    GDD = np.full_like(T_avg, np.nan)

    # Case 1: T_avg <= T_base
    condition_1 = T_avg <= T_base
    GDD[condition_1] = 0

    # Case 2: T_avg > T_base
    condition_2 = T_avg > T_base
    GDD[condition_2] = (T_avg[condition_2] - T_base)*dt

    return GDD

```

```

def gdd_method_2_vect(T_avg, T_base, T_opt, T_upper, dt=1):
    """Vectorized function for computing GDD using method 2"""

    # Pre-allocate the GDD array with NaNs
    GDD = np.full_like(T_avg, np.nan)

    # Case 1: T_avg <= T_base
    condition_1 = T_avg <= T_base
    GDD[condition_1] = 0

    # Case 2: T_base < T_avg <= T_opt
    condition_2 = (T_avg > T_base) & (T_avg <= T_opt)
    GDD[condition_2] = (T_avg[condition_2] - T_base) * dt

    # Case 3: T_opt < T_avg <= T_upper
    condition_3 = (T_avg > T_opt) & (T_avg <= T_upper)
    GDD[condition_3] = ((T_upper-T_avg[condition_3]) / (T_upper-T_opt) * (T_opt-T_base)) *

    # Case 4: T_avg > T_upper
    condition_4 = T_avg > T_upper
    GDD[condition_4] = 0

    return GDD

```

 Tip

In the previous functions, we have opted to pre-allocate an array with NaNs (using `np.full_like(T_avg, np.nan)`) to clearly distinguish between unprocessed and processed data. However, it's also possible to pre-allocate an array of zeros (using `np.zeros_like(T_avg)`). This approach would automatically handle cases 1 and 4 (in method 2), where conditions result in zero values. By doing so, we reduce the number of conditional checks required, making the functions shorter. This choice is particularly useful when zeros accurately represent the desired outcome for certain conditions, contributing to more efficient and concise code.

```

# Test that functions are working as expected
T_avg = np.array([0,12,20,30,40])

print(gdd_method_1_vect(T_avg, T_base))
print(gdd_method_2_vect(T_avg, T_base, T_opt, T_upper))

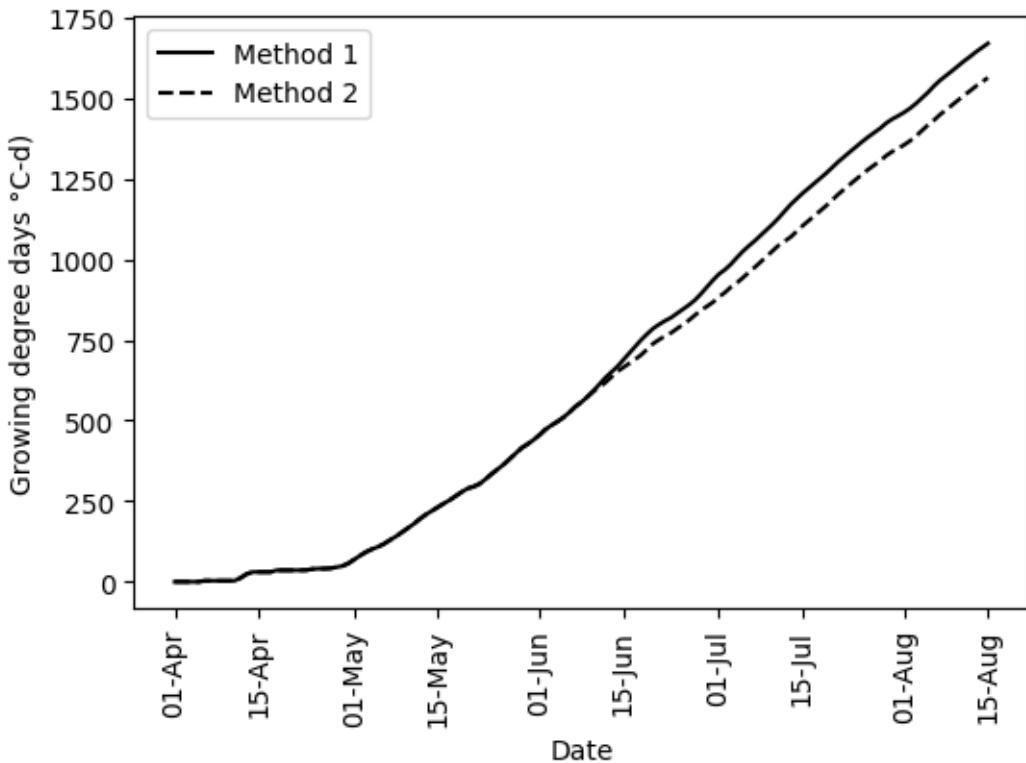
```

```
[ 0  2 10 20 30]  
[ 0  2 10 14  0]
```

```
# Compute GDD  
df_season['GDD_1_vect'] = gdd_method_1_vect(df_season['TEMP2MAVG'], T_base)  
df_season['GDD_2_vect'] = gdd_method_2_vect(df_season['TEMP2MAVG'], T_base, T_opt, T_upper)  
  
df_season['GDD_1_vect_cum'] = df_season['GDD_1'].cumsum()  
df_season['GDD_2_vect_cum'] = df_season['GDD_2'].cumsum()  
  
# Display resulting dataframe (new columns are at the end)  
df_season.head(3)
```

|   | TIMESTAMP  | STATION | PRESSUREAVG | PRESSUREMAX | PRESSUREMIN | SLPAVG | TEMP  |
|---|------------|---------|-------------|-------------|-------------|--------|-------|
| 0 | 2018-04-01 | Gypsum  | 97.48       | 98.18       | 96.76       | 101.93 | 8.70  |
| 1 | 2018-04-02 | Gypsum  | 97.94       | 98.23       | 97.46       | 102.62 | -2.30 |
| 2 | 2018-04-03 | Gypsum  | 96.42       | 97.55       | 95.47       | 101.01 | 1.34  |

```
# Create figure using vectorized columns  
plt.figure(figsize=(6,4))  
  
plt.plot(df_season['TIMESTAMP'], df_season['GDD_1_vect_cum'], '-k', label='Method 1')  
plt.plot(df_season['TIMESTAMP'], df_season['GDD_2_vect_cum'], '--k', label='Method 2')  
plt.xlabel('Date')  
plt.xticks(rotation=90)  
plt.ylabel(f'Growing degree days {chr(176)}C-d')  
plt.legend()  
fmt = mdates.DateFormatter('%d-%b')  
plt.gca().xaxis.set_major_formatter(fmt)  
  
plt.show()
```



### 44.3 Practice

- Search in the provided references or other articles in the literature for alternative methods to compute growing degree days and implement them in Python.
- Merge the code for different methods into a single function. Add an input named `method=` that will allow you to specify which computation method you want to use.
- Convert the non-vectorized functions into vectorized versions using the Numpy function `np.vectorize()`. This option is a convenient way of vectorizing functions, but it's not intended for efficiency. Then, compute the time it takes to compute GDD with each function implementation (non-vectorized, vectorized using Numpy booleans, and vectorized using `np.vectorize()`). Which is one is faster? Which one is faster to code? Hint: For timing the functions use the `perf_counter()` method the time module.

#### **44.4 References**

- McMaster, G.S. and Wilhelm, W., 1997. Growing degree-days: one equation, two interpretations. Agricultural and Forest Meteorology 87 (1997) 291-300
- Nielsen, D. C., & Hinkle, S. E. (1996). Field evaluation of basal crop coefficients for corn based on growing degree days, growth stage, or time. Transactions of the ASAE, 39(1), 97-103.
- Zhou, G. and Wang, Q., 2018. A new nonlinear method for calculating growing degree days. Scientific reports, 8(1), pp.1-14.

# 45 Central dogma

The central dogma of molecular biology explains the flow of genetic information within a biological system. At its core, it involves the process of transcription and translation. In transcription, the genetic code in DNA is transcribed into messenger RNA (mRNA). This mRNA is then used in translation to create polypeptides, which are chains of aminoacids that form proteins. Critical to this process are codons, which are sequences of three nucleotides in the mRNA. Each codon corresponds to a specific aminoacid or a signal to start or stop the translation process by ribosomes. This elegant system of codons ensures that genetic information is accurately translated into the vast array of proteins essential for life.

In this exercise we will create a set of functions to decode DNA sequences with multiple polypeptides. Basically a sort of digital ribosome.

```
# Import modules
import pandas as pd
import pprint

# Load data
lookup = pd.read_csv('../datasets/codon_aminoacids.csv')
lookup.head()
```

|   | codon | letter | aminoacid  |
|---|-------|--------|------------|
| 0 | AAA   | K      | Lysine     |
| 1 | AAC   | N      | Asparagine |
| 2 | AAG   | K      | Lysine     |
| 3 | AAU   | N      | Asparagine |
| 4 | ACA   | T      | Threonine  |

## 45.1 Transcription

Given a sequence of DNA bases we need to find the complementary strand. The catch here is that we also need to account for the fact that the base **thymine** is replaced by the base **uracil** in RNA.

To check for potential typos in the sequence of DNA or to prevent that the user feeds a sequence of mRNA instead of DNA to the transcription function, we will use the `raise` statement, which will automatically stop and exit the `for` loop and throw a custom error message if the code finds a base a base other than A,T,C, or G. The location of the `raise` statement is crucial since we only want to trigger this action if a certain condition is met (i.e. we find an unknown base). So, we will place the `raise` statement inside the `if` statement within the `for` loop. We will also return the location in the sequence of the unknown base using the `find()` method.

The error catching method described above is simple and practical for small applications, but it has some limitations. For instance, we cannot identify whether there are more than one unknown bases and we cannot let the user know the location of all these bases. Nonetheless, this is a good starting point.

```
def transcription(DNA):
    """
    Function that converts a sequence of DNA bases into messenger RNA
    Input: string of DNA
    Author: Andres Patrignani
    Date: 3-Feb-2020
    """

    # Translation table
    transcription_table = DNA.maketrans('ATCG','UAGC')
    #print(transcription_table) {65: 85, 84: 65, 67: 71, 71: 67}

    # Translate using table
    mRNA = DNA.translate(transcription_table)
    return mRNA
```

## 45.2 Translation

The logic of the translation function will be similar to our previous example. The only catch is that we need to keep track of the different polypeptides and the `start` and `stop` signals in the mRNA. These signals dictate the sequence of aminoacids for each polypeptide. Here are some steps of the logic:

- Scan the mRNA in steps of three bases
- Trigger a new polypeptide only when we find the starting ‘AUG’ codon
- After that we know the ribosome is inside the mRNA that encodes aminoacids

- The end of the polypeptide occurs when the ribosome finds any of the stop codons: ‘UAA’, ‘UAG’, ‘UGA’

```
# Translation function

def translation(mRNA):
    """
    Function that decodes a sequence of mRNA into a chain of aminoacids
    Input: string of mRNA
    Author: Andres Patrignani
    Date: 27-Dec-2019
    """

    # Initialize variables
    polypeptides = dict() # More convenient and human-readable than creating a list of lists
    start = False # Ribosome outside region of mRNA that encodes aminoacids
    polypeptide_counter = 0 # A counter to name our polypeptides

    for i in range(0,len(mRNA)-2,3):
        codon = mRNA[i:i+3] # Add 3 to avoid overlapping the bases between iterations.
        aminoacid_idx = lookup.codon == codon # Match current codon with all codons in lookup
        aminoacid = lookup.aminoacid[aminoacid_idx].values[0]

        # Logic to find in which polypeptide the Ribosome is in
        if codon == 'AUG':
            start = True
            polypeptide_counter += 1
            polypeptide_name = 'P' + str(polypeptide_counter)
            polypeptides[polypeptide_name] = []

        elif codon == 'UAA' or codon == 'UAG' or codon == 'UGA':
            start = False

        # If the Ribosome found a starting codon (Methionine)
        if start:
            polypeptides[polypeptide_name].append(aminoacid)

    return polypeptides
```

In the traslation function we could have used `if aminoacid == 'Methionine':` for the first logical statement and `elif aminoacid == 'Stop':` for the second logical statement. I decided to use the codons rather than the aminoacids to closely match the mechanics of the

Ribosome, but the statements are equivalent in terms of the outputs that the function generates.

Q: What happens if you indent four additional spaces the line: `return polypeptide` in the translation function? You will need to modify, save, and call the function to see the answer to this question.

```
DNA = 'TACTCGTCACAGGTTACCCAAACATTTACTGCGACGTATAAACTTACTGCACAAATGTGACT'
mRNA = transcription(DNA)
print(mRNA)
polypeptides = translation(mRNA)
pprint.pprint(polypeptides)
```

```
AUGAGCAGUGUCCAUGGGGUUUGUAAAUGACGCUGCAUAUUUGAAUGACGUGUUUACACUGA
{'P1': ['Methionine',
        'Serine',
        'Serine',
        'Valine',
        'Glutamine',
        'Tryptophan',
        'Glycine',
        'Leucine'],
  'P2': ['Methionine', 'Threonine', 'Leucine', 'Histidine', 'Isoleucine'],
  'P3': ['Methionine', 'Threonine', 'Cysteine', 'Leucine', 'Histidine']}
```

## 46 Frontier production functions

A frontier production function is a method of economic analysis used to assess the use efficiency of a limited number of resources in industry. Perhaps, the most traditional example involves the analysis of resources such as labor and capital required to produce a specific good. The concept has also been widely applied in agriculture to assess the regional (e.g. county, shire) or farm production efficiency as a function of growing season water supply.

In this exercise you will learn how to fit a frontier production function using a dataset of winter wheat grain yield and growing season precipitation for Grant county in Oklahoma. We will implement the Cobb-Douglas model, which is one of the most common models. If you are interested in this topic you should read the seminal paper by Cobb and Douglas published in 1928.

$$y = a + b \ln(x) + c [\ln(x)]^2 , \quad x > 0$$

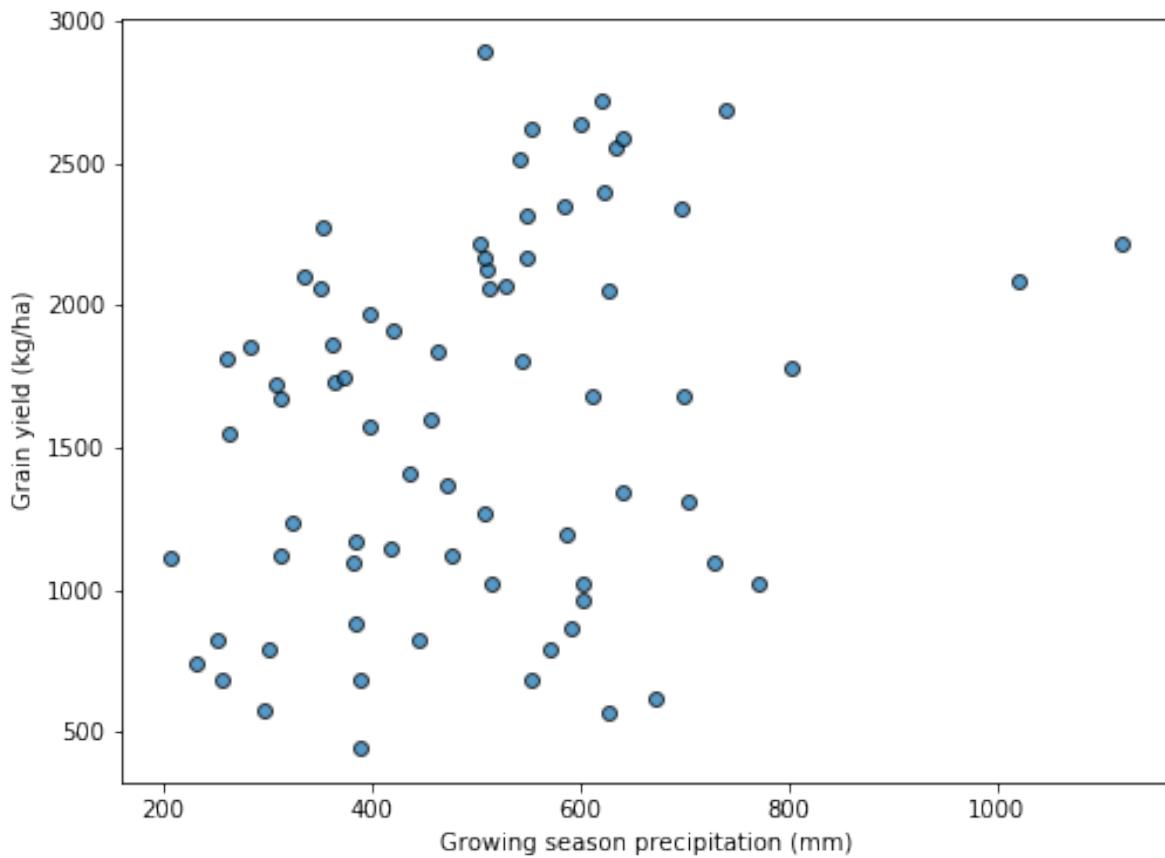
```
# Import modules
import glob
import pandas as pd
import numpy as np
from scipy.optimize import curve_fit, newton
import matplotlib.pyplot as plt

np.seterr(divide = 'ignore'); # Suppress warning about diving by zero

# Load yield and precipitation data
data = pd.read_csv("../datasets/frontier_yield_rainfall.csv", header=1)
data.head()
```

|   | year | county | grain_yield | rainfall |
|---|------|--------|-------------|----------|
| 0 | 1919 | GRANT  | 1021.596244 | 602.107  |
| 1 | 1920 | GRANT  | 1122.411663 | 477.647  |
| 2 | 1922 | GRANT  | 678.823820  | 389.128  |
| 3 | 1923 | GRANT  | 860.291574  | 591.185  |
| 4 | 1924 | GRANT  | 1021.596244 | 516.001  |

```
# Explore data
plt.figure(figsize=(8,6))
plt.scatter(data["rainfall"], data["grain_yield"], edgecolor='k', alpha=0.75)
plt.xlabel('Growing season precipitation (mm)')
plt.ylabel('Grain yield (kg/ha)')
plt.show()
```



Our next step is to partition the dataset into continuous and non-overlapping rainfall intervals that we will call bins, similar to histogram bins.

To generate the bins we will search for the minimum and maximum values of the rainfall vector and then use this information to generate linearly spaced (it could also be log-spaced) rainfall values that we will use to partition the dataset at regular intervals. In other words, each bin will be defined by a lower and upper rainfall boundary.

Then, the idea is to select the highest observed grain yield obtained within a specific bin. We will also need to identify the position of the highest yield observation for that specific bin

because we also need to retrieve the growing season rainfall at which the highest yield was achieved. The highest yield and its corresponding rainfall within a bin represent a pair of  $x, y$  coordinates that we can use to approximate the frontier function. Of course we will collect several of these pairs, as many as  $N-1$ , where  $N$  denotes the number of bins.

As usual, we need to consider potentially missing values (represented as NaN) or perhaps rainfall bins that do not contain any yield data. The latter situation could arise from a dataset with insufficient yield observations in certain rainfall ranges or as a result of dividing the dataset into a large number of bins, and therefore, so narrow that there might not be any historical yield record for that specific narrow bin.

```
# Identify boundary points
max_rainfall = np.max(data.rainfall)
min_rainfall = np.min(data.rainfall)
N = 15 # Number of bins
rainfall_bins = np.linspace(min_rainfall, max_rainfall, N)
print(rainfall_bins)
```

```
[ 206.629      271.80721429  336.98542857  402.16364286  467.34185714
 532.52007143  597.69828571  662.8765      728.05471429  793.23292857
 858.41114286  923.58935714  988.76757143 1053.94578571 1119.124      ]
```

Now that we have the intervals for our rainfall bins it is time to test whether we can select rainfall events within the lower and upper bins limits. We will test this operation by hard coding a lower and upper limit. I will make the arbitrary choice of select **great or equal than** for the lower limit and **greater than** for the upper limit.

Note how we are breaking down the problem and testing individual components of a larger operation. I highly encourage you to build your code line by line. Test the code using a simple example to ensure that it works as expected. Then, proceed with another line.

```
# Let's test the boolean for selecting rainfall values above the bin lower limit
# Run this cell to see the list of True and False entries (it is long)
data.rainfall >= rainfall_bins[0]
```

```
0    True
1    True
2    True
3    True
4    True
...
...
```

```
68    True
69    True
70    True
71    True
72    True
Name: rainfall, Length: 73, dtype: bool
```

```
# Let's test the boolean for selecting rainfall values below the bin upper limit
# Run this cell to see the list of True and False entries (it is long)
data.rainfall < rainfall_bins[1]
```

```
0    False
1    False
2    False
3    False
4    False
...
68   False
69   False
70   False
71   False
72   False
Name: rainfall, Length: 73, dtype: bool
```

What we really need to do is to put these two statements together. Only values that are `True` in both statements will result to be also `True` in the final statement.

```
# Let's test these two boolean statements together for n = 1
n = 1
(data.rainfall >= rainfall_bins[n]) & (data.rainfall < rainfall_bins[n+1])
```

```
0    False
1    False
2    False
3    False
4    False
...
68   False
69   False
70   False
```

```

71    False
72    True
Name: rainfall, Length: 73, dtype: bool

```

We have the bins and we know how to select years (because rainfall and yield are provided for different years) within different rainfall intervals.

The next step consists of iterating over each bin, get the highest yield for the bin, get the matching growing season rainfall, store these value, and proceed with the next bin until we exhaust all the bins (i.e. we reach the maximum rainfall value)

As mentioned earlier, it will be wise to account for bins with no yield, just in case we divided the rainfall vector in way too many intervals. I will simple check if all the boolean values are **False**. Our code may still crash due to other issues that aren't within our radar.

What can I do to anticipate errors or roadblocks in my code? First I suggest exploring your data. Open and inspect the file to check for missing values, mixed strings and numbers, etc. In addition, I highly recommend sketching the method by drawing a chart. You will need to instruct the computer to follow the same steps. So drawing and sketching are excellent exercises to breakdown problems into smaller steps.

```

# Loop over each rainfall interval

# Initiate the empty array to collect values
frontier_yield_obs = np.array([])
frontier_rainfall_obs = np.array([])

# The loop below will go to N-1, so that when we reach the end of the loop
# and we call N+1 we retrieve the last element. Otherwise, if we iterate until N
# when we call the N+1 element it will return an error since there is no such element.
for n in range(0,len(rainfall_bins)-1):
    idx = (data.rainfall >= rainfall_bins[n]) & (data.rainfall < rainfall_bins[n+1])

    if np.all(idx == False):
        continue

    else:
        rainfall_bin = data.loc[idx, 'rainfall']
        yield_bin = data.loc[idx, 'grain_yield']

        max_yield_bin = np.amax(yield_bin)
        idx_max_yield_bin = np.argmax(yield_bin.values)

```

```

corresponding_rainfall_bin = rainfall_bin.iloc[idx_max_yield_bin]

frontier_rainfall_obs = np.append(frontier_rainfall_obs, corresponding_rainfall_bin)
frontier_yield_obs = np.append(frontier_yield_obs, max_yield_bin)

# Display outputs. For simplicity I rounded the outputs at the time of printing.
print(np.round(frontier_rainfall_obs))
print(np.round(frontier_yield_obs))

[ 261.  336.  354.  422.  509.  553.  621.  697.  739.  803. 1020.]
[1815. 2104. 2272. 1915. 2890. 2621. 2722. 2339. 2688. 1781. 2084.]

```

Done! We have the highest yield and the corresponding rainfall for each selected yield. Note how the variable names, despite being a bit verbose, actually help to read and follow the logic of the code.

The last step to create the frontier production function consists of defining the model of the frontier and optimizing model parameters based the pairwise rainfall-yield data that we just collected for the rainfall bins.

There are multiple Python packages that can be used to optimize functions. We will use the standard `curve_fit` function based on non-linear least squares from the SciPy module.

```

# Define Cobb-Douglas model
cobb_douglas = lambda x, a, b, c: a + b * np.log(x) + c * np.log(x)**2

# Define initial guess for the parameters of the Cobb-Douglas model
par0 = [1,1,1]

# Fit Cobb-Douglas model
par = curve_fit(cobb_douglas, frontier_rainfall_obs, frontier_yield_obs, par0)
print('Optimized parameters:', np.round(par[0]))

# Create range of x values using a step increase (default). This will give us enough detail
# to have an accurate curve for most practical purposes.
frontier_rainfall_line = np.arange(0,max_rainfall)
frontier_yield_line = cobb_douglas(frontier_rainfall_line, *par[0])

print('The frontier was built using a total of:', len(frontier_rainfall_line), 'points')

```

```

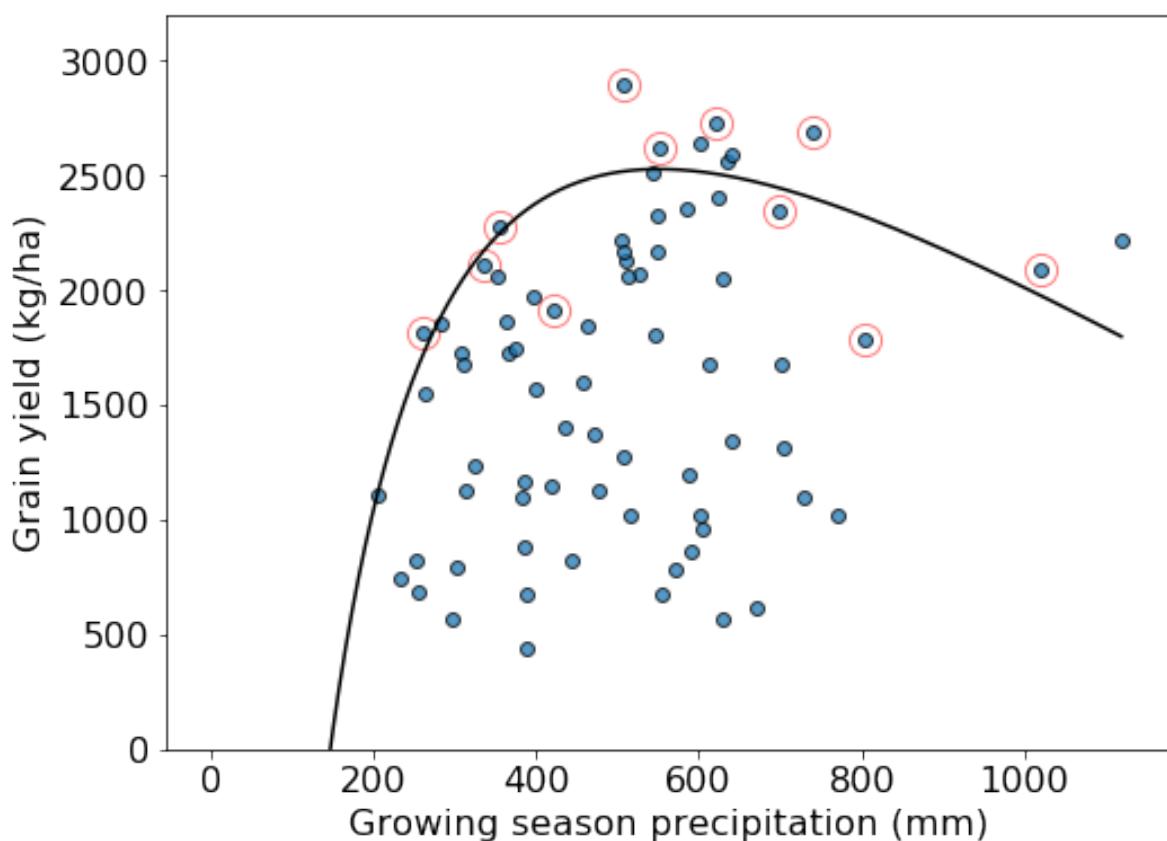
Optimized parameters: [-55245.  18306. -1450.]
The frontier was built using a total of: 1120 points

```

```

# Plot the data and frontier
plt.figure(figsize=(8,6))
plt.scatter(frontier_rainfall_obs, frontier_yield_obs, s=200, alpha=0.5, facecolor='w', color='black')
plt.scatter(data["rainfall"], data["grain_yield"], edgecolor='k', alpha=0.75)
plt.plot(frontier_rainfall_line, frontier_yield_line, '-k')
plt.xlabel('Growing season precipitation (mm)', size=16)
plt.ylabel('Grain yield (kg/ha)', size=16)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.ylim(0,3200)
plt.show()

```



## 46.1 Curve interpretation

- The part of the frontier intercepting the y axis is negative. We restricted the plot to plausible yield values, which are certainly greater than zero.

- The part of the frontier that intercepts the `x` axis represents the amount of growing season water supply that generates zero yield. This can be viewed as an inefficiency of the system and represents the minimum water losses (due to runoff, evaporation, drainage, canopy interception, etc.).
- The highest point of the curve gives us an idea of the growing season rainfall required to achieve the highest yields. Note that during many years yields in the central range can be much lower than the highest yield. To a great extent this is associated to other factors, such as the distribution of the rainfall during the growing season or the occurrence of other factors like heat stress, hail damage, or yield losses due to diseases and pests.
- The last, and decaying, portion of the curve could be due to two main reasons: 1) There is insufficient yield data for years with high growing season rainfall. The chances of receiving two or three times as much rainfall than the median rainfall in a single growing season are probably not very high. 2) Excess of water can be detrimental to the production of grain yield. This actually makes sense and can be related to flooding events, plant lodging, increased disease pressure, weaker root anchoring, and even larger number of cloudy days that reduce the amount of solar radiation for photosynthesis. We don't know the answer from this dataset.

To complete the analysis we will compute few extra metrics that might be useful to summarize the dataset. I assume most of you are familiar with the concept of median (i.e. 50th percentile), which is a useful metric of central tendency robust to outliers and skewed distributions. Perhaps, the most challenging step is the one regarding the searching of the function roots. We can use the Newton-Raphson method (also known as the secant method) for finding function roots (i.e. at what value(s) of  $x$  the function  $f(x)$  intersects the  $x$ -axis). Based on a visual inspection of the graph, we will pass an initial guess for the search of 200 mm. If you want to learn more about this function you can read the official [SciPy documentation](#).

We can also estimate the ideal rainfall by finding the point at which the frontier does not show any additional increase in yield with increasing rainfall. We can find this optimum point by finding the point at which the first derivative of the frontier function is zero.

## 46.2 Additional summary metrics

```
# Median rainfall
median_rainfall = np.median(data.rainfall)
print(median_rainfall, 'mm')

# Median grain yield
median_yield = np.median(data.grain_yield)
print(median_yield, 'kg/ha')
```

```

# Estimate minimum_losses using the Newton-Raphson method
minimum_losses = newton(cobb_douglas, 200, args=par[0])
print('Minimum losses: ', round(minimum_losses), 'mm')

# Optimal rainfall (system input) and grain yield (system output)
# We will approximate the first derivative using the set of points (i.e. numerical approximation)
# Step 1: Calculate derivative, Step 2: Calculate absolute value, Step 3: find minimum value
first_diff_approax = np.diff(frontier_yield_line)
idx_zero_diff = np.argmin(np.abs(first_diff_approax))
optimal_rainfall = frontier_rainfall_line[idx_zero_diff]
optimal_yield = frontier_yield_line[idx_zero_diff]
print('Optimal rainfall:', round(optimal_rainfall), 'mm')
print('Optimal yield:', round(optimal_yield), 'kg/ha')

```

```

508.254 mm
1680.25698 kg/ha
Minimum losses: 147.0 mm
Optimal rainfall: 550.0 mm
Optimal yield: 2525.0 kg/ha

```

## 46.3 Quantile regression

In the previous line of reasoning we have focused on selecting the highest yield within a given rainfall interval. This approach makes direct use of both rainfall and yield observations to build the frontier production function.

An alternative approach is to calculate some statistical variables for each interval. A commonly used technique is that of selecting percentiles or quantiles. You probably heard the term “Quantile regression analysis”. I’m sure that Python and R packages have some advanced features, but here I want show the concept behind the technique, which is similar to the approach described earlier.

For each bin we will simply calculate the yield 95th percentile and the average rainfall. The pairwise points will not necessarily match observations and will likely be smoother since we are filtering out some of the noise.

```

# Quantile regression
max_rainfall = np.max(data.rainfall)
min_rainfall = np.min(data.rainfall)
N = 10 # Number of bins

```

```

rainfall_bins = np.linspace(min_rainfall, max_rainfall, N)

frontier_yield_obs = np.array([])
frontier_rainfall_obs = np.array([])

for n in range(0,len(rainfall_bins)-1):
    idx = (data.rainfall >= rainfall_bins[n]) & (data.rainfall < rainfall_bins[n+1])

    if np.all(idx == False):
        continue

    else:
        rainfall_bin = data.loc[idx, 'rainfall']
        yield_bin = data.loc[idx, 'grain_yield']

        frontier_rainfall_obs = np.append(frontier_rainfall_obs, np.mean(rainfall_bin))
        frontier_yield_obs = np.append(frontier_yield_obs, np.percentile(yield_bin, 95))

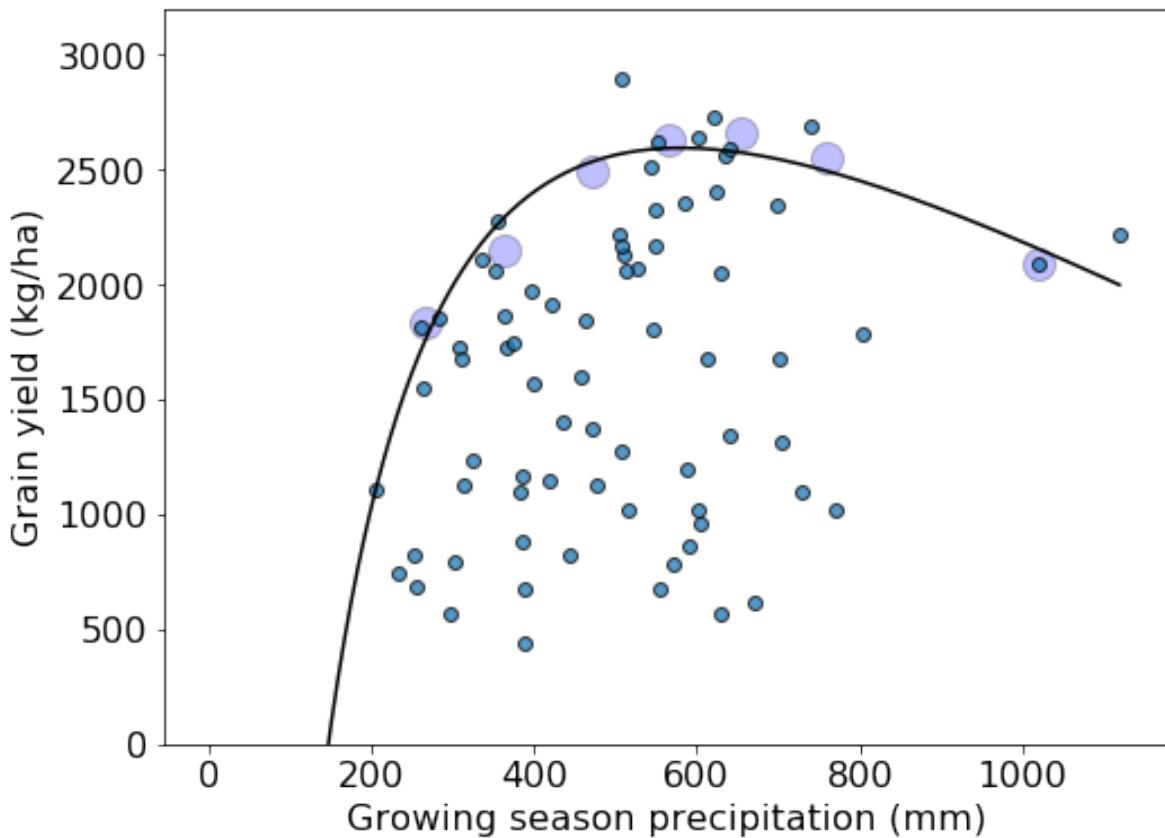
par0 = [1,1,1]

par = curve_fit(cobb_douglas, frontier_rainfall_obs, frontier_yield_obs, par0)

frontier_rainfall_line = np.arange(0,max_rainfall)
frontier_yield_line = cobb_douglas(frontier_rainfall_line, *par[0])

# Plot the data and frontier
plt.figure(figsize=(8,6))
plt.scatter(frontier_rainfall_obs, frontier_yield_obs,
            s=200, alpha=0.25, facecolor='b', edgecolors='k', linewidths=1)
plt.scatter(data["rainfall"], data["grain_yield"], edgecolor='k', alpha=0.75)
plt.plot(frontier_rainfall_line, frontier_yield_line, '-k')
plt.xlabel('Growing season precipitation (mm)', size=16)
plt.ylabel('Grain yield (kg/ha)', size=16)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.ylim(0,3200)
plt.show()

```



```

# Estimate minimum_losses using the Newton-Raphson method
minimum_losses = newton(cobb_douglas, 200, args=par[0])
print('Minimum losses: ', round(minimum_losses), 'mm')

# Optimal rainfall (system input) and grain yield (system output)
# We will approximate the first derivative using the set of points (i.e. numerical approximation)
# Step 1: Calculate derivative, Step 2: Calculate absolute value, Step 3: find minimum value
first_diff_approax = np.diff(frontier_yield_line)
idx_zero_diff = np.argmin(np.abs(first_diff_approax))
optimal_rainfall = frontier_rainfall_line[idx_zero_diff]
optimal_yield = frontier_yield_line[idx_zero_diff]
print('Optimal rainfall:', round(optimal_rainfall), 'mm')
print('Optimal yield:', round(optimal_yield), 'kg/ha')

```

```

Minimum losses: 147.0 mm
Optimal rainfall: 580.0 mm
Optimal yield: 2592.0 kg/ha

```

## 46.4 Observations

Depending on the method we obtained somewhat different values of minimum losses, optimal rainfall, and optimal yield. So here are some questions for you to think:

- As a researcher how do we determine the right method to analyze our data? Particularly when methods result in slightly different answers.
- Run the code again using a different number of bins. How different are the values for minimum losses and optimum rainfall amounts?
- Should we consider an asymptotic model or a model like the Cobb-Douglas that may exhibit a decreasing trend at high growing season rainfall amounts?

## 46.5 References

Cobb, C.W. and Douglas, P.H., 1928. A theory of production. *The American Economic Review*, 18(1), pp.139-165.

French, R.J. and Schultz, J.E., 1984. Water use efficiency of wheat in a Mediterranean-type environment. I. The relation between yield, water use and climate. *Australian Journal of Agricultural Research*, 35(6), pp.743-764.

Grassini, P., Yang, H. and Cassman, K.G., 2009. Limits to maize productivity in Western Corn-Belt: a simulation analysis for fully irrigated and rainfed conditions. *Agricultural and forest meteorology*, 149(8), pp.1254-1265.

Patrignani, A., Lollato, R.P., Ochsner, T.E., Godsey, C.B. and Edwards, J., 2014. Yield gap and production gap of rainfed winter wheat in the southern Great Plains. *Agronomy Journal*, 106(4), pp.1329-1339.

## 47 Soil Water Storage

Example for calculation of soil water storage using soil moisture data recorded at two different dates and at ten different soil depths using a neutron probe in a wheat field located near Lahoma, OK. The goal is to integrate the soil moisture values along the soil profile for each date to calculate the soil water storage.

We will then subtract the soil water storage from two consecutive dates to calculate the change in soil water storage.

```
# Import modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Read file with soil moisture data
df = pd.read_csv('../datasets/profile_soil_moisture_lahoma_ntw.csv')
df.head()
```

|   | profile_layer | 7/2/2009 | 7/10/2009 | 7/16/2009 | 7/24/2009 | 7/30/2009 | 8/7/2009 | 8/12/2009 | 8/21/2009 |
|---|---------------|----------|-----------|-----------|-----------|-----------|----------|-----------|-----------|
| 0 | 0-20          | 0.135    | 0.198     | 0.159     | 0.208     | 0.243     | 0.205    | 0.251     | 0.248     |
| 1 | 20-40         | 0.172    | 0.240     | 0.237     | 0.276     | 0.274     | 0.291    | 0.318     | 0.321     |
| 2 | 40-60         | 0.228    | 0.237     | 0.237     | 0.257     | 0.259     | 0.277    | 0.327     | 0.338     |
| 3 | 60-80         | 0.260    | 0.262     | 0.262     | 0.268     | 0.267     | 0.269    | 0.301     | 0.333     |
| 4 | 80-100        | 0.157    | 0.175     | 0.179     | 0.194     | 0.192     | 0.198    | 0.217     | 0.320     |

```
# Create array with depths
depths = np.arange(0,200,20) # depths in cm
print(depths)
```

```
[ 0  20  40  60  80 100 120 140 160 180]
```

## 47.1 Trapezoidal integration

Before calculating the soil water storage for all dates we will first compute the storage for a single date to ensure our calculations are correct.

The trapezoidal rule is a discrete integration method that basically adds up a collection of trapezoids. The narrower the intervals the more accurate the method, particularly when dealing with sudden non-linear changes.

**Figure:** An animated gif showing how the progressive reduction in step size increases the accuracy of the approximated area below the function. Khurram Wadee (2014). This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

```
vwc_1 = df["7/2/2009"].values # volumetric water content
storage_1 = np.trapz(vwc_1,depths) # total profile soil water storage in cm

print(storage_1,"cm of water in 2-Jul-2009")
```

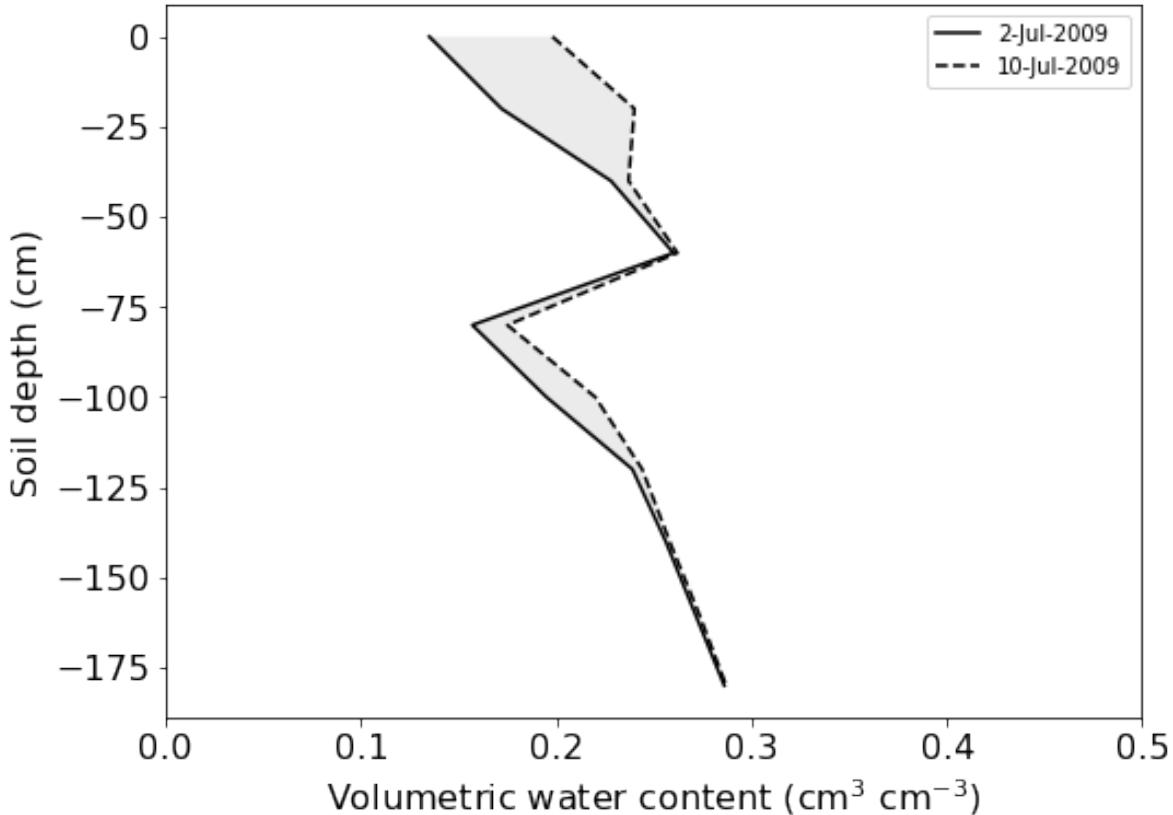
39.77 cm of water in 2-Jul-2009

```
vwc_2 = df["7/10/2009"].values # volumetric water content
storage_2 = np.trapz(vwc_2,depths) # total profile soil water storage in cm

print(storage_2,"cm of water in 10-Jul-2009")
```

43.03 cm of water in 10-Jul-2009

```
# Plot profile
plt.figure(figsize=(8,6))
plt.plot(vwc_1,depths**-1, '-k', label="2-Jul-2009")
plt.plot(vwc_2,depths**-1, '--k', label="10-Jul-2009")
plt.xlabel('Volumetric water content (cm$^3$ cm$^{-3}$)', size=16)
plt.ylabel('Soil depth (cm)', size=16)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.legend()
plt.fill_betweenx(depths**-1, vwc_1, vwc_2, facecolor=(0.7,0.7,0.7), alpha=0.25)
plt.xlim(0,0.5)
plt.show()
```



```
# Compute total soil water storage for each date
storage = np.array([])
for date in range(1,len(df.columns)):
    storage_date = np.round(np.trapz(df.iloc[:,date], depths), 2)
    storage = np.append(storage,storage_date)

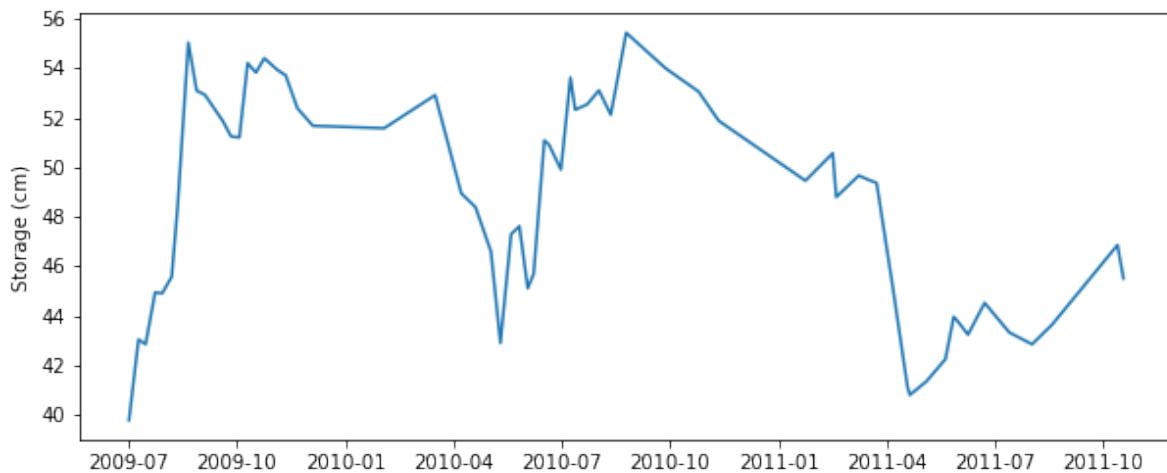
storage
```

```
array([39.77, 43.03, 42.84, 44.93, 44.9 , 45.57, 48.42, 55.03, 53.09,
      52.92, 51.87, 51.24, 51.2 , 54.2 , 53.82, 54.4 , 53.93, 53.71,
      52.37, 51.67, 51.57, 52.91, 48.94, 48.38, 46.59, 42.89, 47.29,
      47.61, 45.1 , 45.69, 51.08, 50.9 , 49.9 , 53.62, 52.32, 52.53,
      53.1 , 52.12, 55.43, 54. , 53.05, 51.87, 49.45, 50.56, 48.79,
      49.66, 49.49, 49.36, 45.03, 41.13, 40.79, 41.34, 42.24, 43.95,
      43.79, 43.23, 44.51, 43.31, 42.84, 43.64, 46.85, 45.5 ])
```

```
# Get measurement dates and convert them to datetime format
obs_dates = pd.to_datetime(df.columns[1:])
obs_delta = obs_dates - obs_dates[0]
obs_seq = obs_delta.days
print(len(obs_seq))
```

62

```
# Plot timeseries of profile soil moisture
plt.figure(figsize=(10,4))
plt.plot(obs_dates, storage)
plt.ylabel('Storage (cm)')
plt.show()
```



```
# Y values
#y = np.tile(depths*-1,62)
y = np.repeat(depths*-1,62)
y.shape
```

(620,)

```
# X values
x = np.tile(obs_seq,10)
x.shape
```

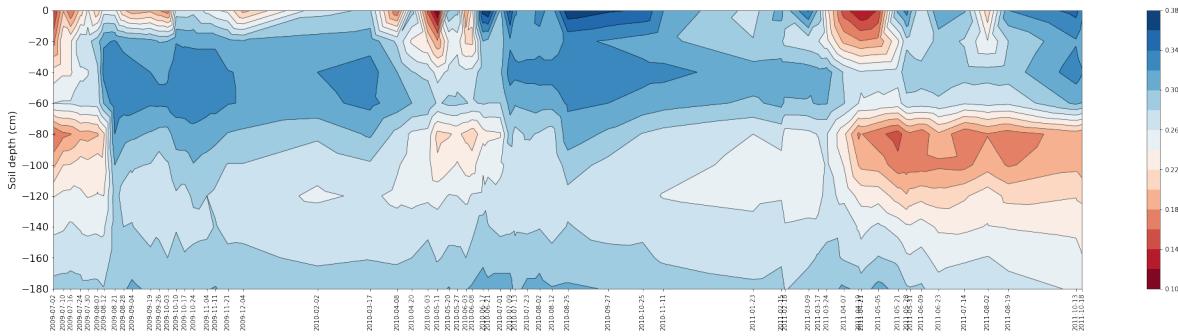
```
(620,)
```

```
# Z values
z = df.iloc[:,1: ].values.flatten()
z.shape
```

```
(620,)
```

## 47.2 Contour plot

```
plt.figure(figsize=(36,8))
plt.tricontour(x, y, z, levels=14, linewidths=0.5, colors='k')
plt.tricontourf(x, y, z, levels=14, cmap="RdBu")
plt.xticks(obs_seq, labels=obs_dates.date, rotation=90)
plt.colorbar(label="Volumetric Water Content")
plt.ylabel('Soil depth (cm)', size=16)
plt.yticks(fontsize=16)
plt.show()
```



## 47.3 References

Patrignani, A., Godsey, C.B., Ochsner, T.E. and Edwards, J.T., 2012. Soil water dynamics of conventional and no-till wheat in the Southern Great Plains. *Soil Science Society of America Journal*, 76(5), pp.1768-1775.

Yimam, Y.T., Ochsner, T.E., Kakani, V.G. and Warren, J.G., 2014. Soil water dynamics and evapotranspiration under annual and perennial bioenergy crops. *Soil Science Society of America Journal*, 78(5), pp.1584-1593.

# 48 Plant Available Water

In the early 1900s, soil physics started to emerge as a prominent discipline with the goal of better understanding the soil-plant-atmosphere continuum. At that time, the knowledge about the energy-state of soil water was transitioning from a qualitative and discrete classification system (e.g. gravitational, capillary, and hygroscopic) to a quantitative and continuous framework.

An upper and lower soil water content was devised to represent the fraction of soil water that is available to plants, particularly agricultural crops. These concepts are known as *Field Capacity* (FC) and *Permanent Wilting Point* (WP) and represent a basic, yet practical, framework that guided irrigation decisions and crop models for many decades. For a specific layer, the plant available water using this approach is computed as:

$$PAW = (\theta_{fc} - \theta_{wp})\Delta z$$

where  $\theta$  represents the volumetric soil water content and  $z$  the thickness of the soil layer under consideration.

In recent years more sophisticated frameworks emerged to address some of the limitations of the previous method to compute available soil water. For instance, the previous method assumes that the same amount of work is required to extract a unit of soil water along the spectrum from FC to PWP. Two methods that emerged as a substitute of the previous method are: 1) Integral water capacity (Groenevelt et al., 2001) and 2) Integral energy (Minasny and McBratney, 2003).

## 48.1 Integral energy

The integral energy approach aims at characterizing the total amount of work required to extract a given amount of water from the soil. This approach can be useful to better understand plant responses to soil water stress since it does not assume equal availability of water between two potentials like the traditional available water capacity approach.

$$E_i = \int_{\theta_i}^{\theta_f} \frac{1}{\theta_i - \theta_f} \psi(\theta) d\theta$$

We will use the soil water retention model proposed by van Genuchten (1980) since it is the most familiar for students in soil science.

We will also use the clay and silty clay soil in the original manuscript published by Minasny and McBratney 2003 as an example. The soil water retention curves of these two soils have similar values of volumetric water content at -10 J/kg and -1500 J/kg, but the concavity between these two points is different, which should result in similar plant available water using the traditional approach, but different amount of work between these two points using the integral energy approach.

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt

# Define soil water retention model (van Genuchten 1980)
model = lambda x,alpha,n,theta_r,theta_s: theta_r+(theta_s-theta_r)*(1+(alpha*x)**n)**-(1-n)

# Range in matric potential
fc = 10    # Field capacity (J/kg)
wp = 1500  # Wilting point (J/kg)
N = 10000

# Define absolute values of matric potential
matric = np.logspace(np.log10(fc), np.log10(wp), N)

# Krasnozem clay (Table 1 Minasny and McBratney, 2003)
theta_clay = model(matric, 1.22, 1.34, 0.23, 0.64)

# Volumeetric water content values at filed capacity and wilting point
fc_clay = theta_clay[0]  # Field capacity, upper limit
wp_clay = theta_clay[-1] # Wilting point, lower limit

# Plant available water
paw_clay = fc_clay - wp_clay

print('Clay at -10 J/kg:', round(fc_clay,3))
print('Clay at -1500 J/kg:', round(wp_clay,3))
print('Clay Plant Available Water Capacity',round(paw_clay,3), "cm^3/cm^3")
```

Clay at -10 J/kg: 0.404  
Clay at -1500 J/kg: 0.262

Clay Plant Available Water Capacity 0.142 cm<sup>3</sup>/cm<sup>3</sup>

```
# Xanthozem silty-clay (Table 1 Minasny and McBratney, 2003)
theta_silty_clay = model(matric, 0.05, 1.1, 0, 0.42)

# Volumetric water content values at field capacity and wilting point
fc_silty_clay = theta_silty_clay[0] # Field capacity, upper limit
wp_silty_clay = theta_silty_clay[-1] # Wilting point, lower limit

# Plant available water
paw_silty_clay = fc_silty_clay - wp_silty_clay

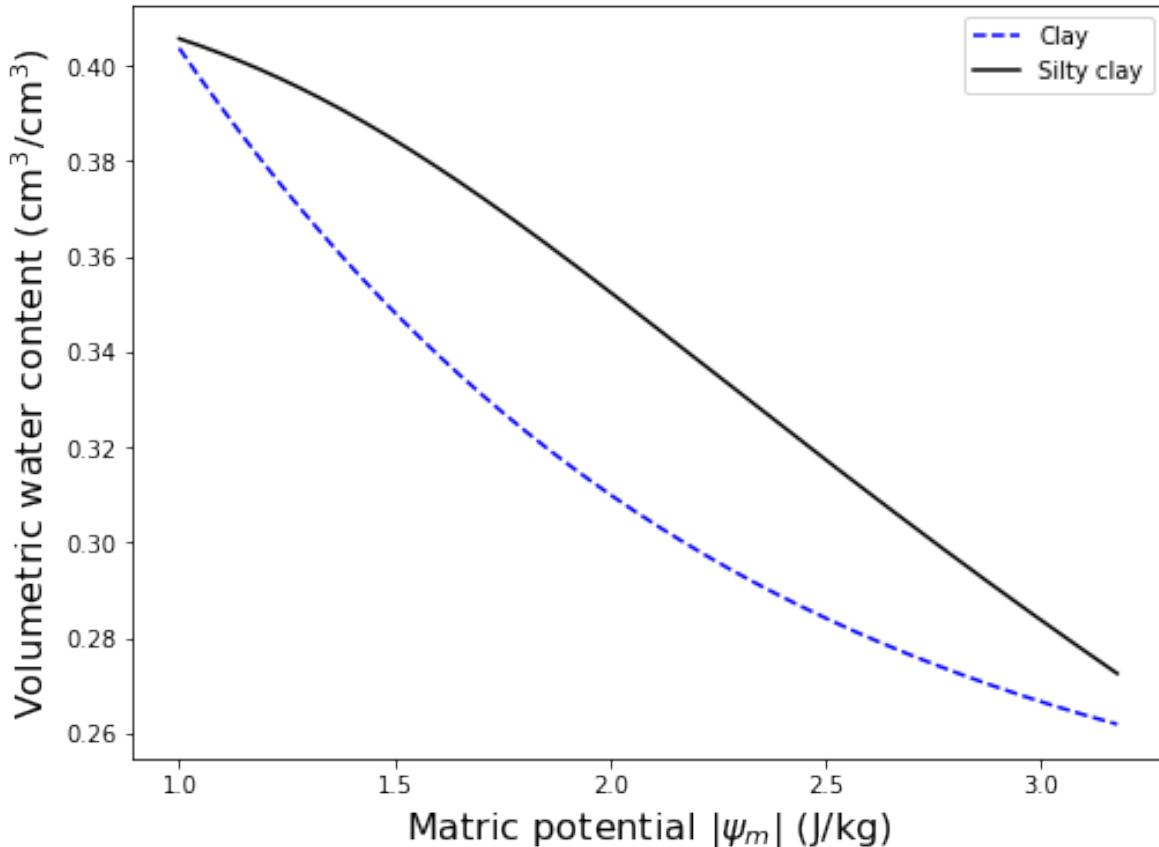
print('Silty-clay at -10 J/kg:', round(fc_silty_clay,3))
print('Silty-clay at -1500 J/kg:', round(wp_silty_clay,3))
print('Silty clay Plant Available Water Capacity',round(paw_silty_clay,3), "cm^3/cm^3")
```

Silty-clay at -10 J/kg: 0.406

Silty-clay at -1500 J/kg: 0.273

Silty clay Plant Available Water Capacity 0.133 cm<sup>3</sup>/cm<sup>3</sup>

```
plt.figure(figsize=(8,6))
plt.plot(np.log10(matric), theta_clay, '--b', label='Clay')
plt.plot(np.log10(matric), theta_silty_clay, '-k', label='Silty clay')
plt.xlabel("Matric potential $|\psi_m|$(J/kg)", size=16)
plt.ylabel("Volumetric water content (cm$^3$/cm$^3$)", size=16)
plt.legend()
plt.show()
```



## 48.2 Soil water storage

```
# Total storage clay between -10 and -1500 J/kg
W_clay = 1/np.abs(fc - wp) * np.trapz(theta_clay, matric)
print(W_clay)
```

0.2768349634688867

```
# Total storage silty clay between -10 and -1500 J/kg
storage_silty_clay = 1/np.abs(fc - wp) * np.trapz(theta_silty_clay, matric)
print(storage_silty_clay)
```

0.3001711916723065

## 48.3 Energy

Because  $\theta$  in our code is in decreasing order, the  $\Delta x$  during the trapezoidal integration will result in negative values. So, I swapped the minuend and subtrahend from  $\theta_i - \theta_f$  to  $\theta_f - \theta_i$  to reverse the sign.

```
# Integral energy
theta_i = theta_clay[0]
theta_f = theta_clay[-1]
E_clay = 1/(theta_f - theta_i) * np.trapz(matric, x=theta_clay)
print(round(E_clay**-1), 'J/kg are required to go from FC to WP')
```

-167.0 J/kg are required to go from FC to WP

```
theta_i = theta_silty_clay[0]
theta_f = theta_silty_clay[-1]
E_silty_clay = 1/(theta_f - theta_i) * np.trapz(matric, x=theta_silty_clay)
print(round(E_silty_clay**-1), 'J/kg are required to go from FC to WP')
```

-319.0 J/kg are required to go from FC to WP

## 48.4 References

Groenevelt, P.H., Grant, C.D. and Semetsa, S., 2001. A new procedure to determine soil water availability. *Soil Research*, 39(3), pp

Hendrickson, A.H. and Veihmeyer, F.J., 1945. Permanent wilting percentages of soils obtained from field and laboratory trials. *Plant physiology*, 20(4), p.517.

Minasny, B. and McBratney, A.B., 2003. Integral energy as a measure of soil-water availability. *Plant and Soil*, 249(2), pp.253-262.

Veihmeyer, F.J. and Hendrickson, A.H., 1931. The moisture equivalent as a measure of the field capacity of soils. *Soil Science*, 32(3), pp.181-194.

## 49 Photoperiod

The photoperiod (i.e. daylight hours) is a variable highly related to plant development, particularly regulating the transition between vegetative and reproductive stages, a transition that is typically characterized by flowering. This is an evolutionary adaptation to ensure that seed production occurs during the right environmental conditions to increase the survival rate and ensure the perpetuation of the species.

The length of light period required to induce flowering depends on plant species. Some plants like barley and wheat are induced to flowering when days are becoming longer (known as long-day plants), while some plants like cotton, rice, and chrysanthemums change stages when days are becoming shorter (known as short-day plants).

While initially researchers thought it was the number of daylight hours that influenced the physiological and morphological changes in plants, it was later discovered that it is actually the length of dark hours that regulates plant development. So, short-day plants are actually long-night plants, and viceversa.

Photoperiodism also affects animals by affecting migration patterns, triggering the entry into hibernation, and conditioning sexual behaviour.

Because of its tilted angle and the consistent orbit of the Earth around the sun, the theoretical photoperiod can be accurately estimated based on the day of the year and latitude. Of course, the effective photoperiod observed at the Earth's surface may change depending on sky conditions.

The goal of this challenge is to write a Python script to compute the number of daylight hours for a **user-specified** date and latitude.

You can learn more about photoperiodism at:

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt

# Define function
def photoperiod(phi,doy,verbose=False):
```

```

phi = np.radians(phi) # Convert to radians
light_intensity = 2.206 * 10***-3

C = np.sin(np.radians(23.44)) # sin of the obliquity of 23.44 degrees.
B = -4.76 - 1.03 * np.log(light_intensity) # Eq. [5]. Angle of the sun below the horizon

# Calculations
alpha = np.radians(90 + B) # Eq. [6]. Value at sunrise and sunset.
M = 0.9856*doy - 3.251 # Eq. [4].
lmd = M + 1.916*np.sin(np.radians(M)) + 0.020*np.sin(np.radians(2*M)) + 282.565 # Eq. [1]
delta = np.arcsin(C*np.sin(np.radians(lmd))) # Eq. [2]. 

# Defining sec(x) = 1/cos(x)
P = 2/15 * np.degrees( np.arccos( np.cos(alpha) * (1/np.cos(phi)) * (1/np.cos(delta)) )

# Print results in order for each computation to match example in paper
if verbose:
    print('Input latitude =', np.degrees(phi))
    print('[Eq 5] B =', B)
    print('[Eq 6] alpha =', np.degrees(alpha))
    print('[Eq 4] M =', M[0])
    print('[Eq 3] Lambda =', lmd[0])
    print('[Eq 2] delta=', np.degrees(delta[0]))
    print('[Eq 1] Daylength =', P[0])

return P

# Invoke function with scalars
phi = 33.4; # Latitude for consistency with notation in literature.
doy = np.array([201]); # Day of the year. Julian calendar. Day from January 1.

P = photoperiod(phi,doy,verbose=True)
print('Photoperiod: ' + str(np.round(P[0],2)) + ' hours/day')

```

```

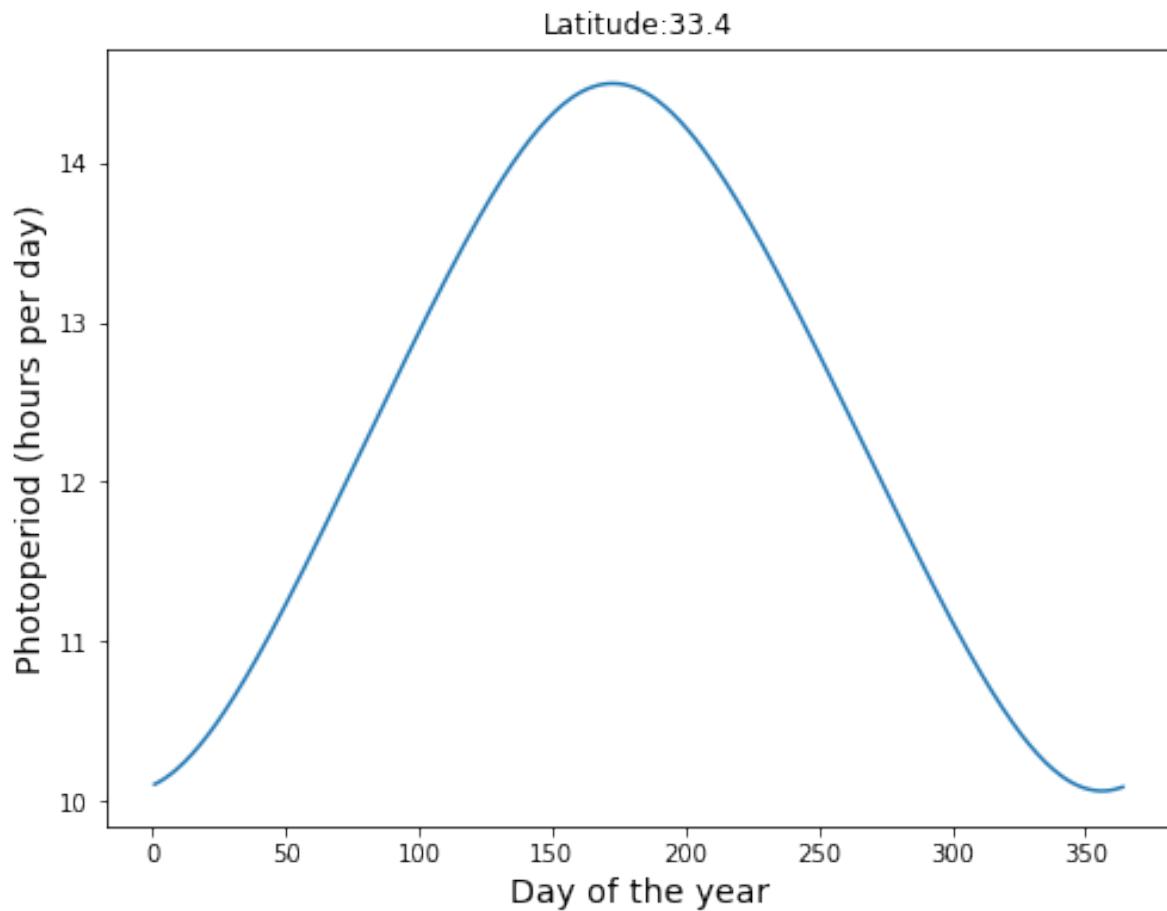
Input latitude = 33.4
[Eq 5] B = 1.5400715888953513
[Eq 6] alpha = 91.54007158889536
[Eq 4] M = 194.8546
[Eq 3] Lambda = 476.93831283687416
[Eq 2] delta= 20.770548026002125
[Eq 1] Daylength = 14.203998218048154

```

Photoperiod: 14.2 hours/day

```
# Multiple inputs call
phi = 33.4;
doy = np.arange(1,365);
P = photoperiod(phi,doy)

plt.figure(figsize=(8,6))
plt.plot(doy,P)
plt.title('Latitude:' + str(phi))
plt.xlabel('Day of the year', size=14)
plt.ylabel('Photoperiod (hours per day)', size=14)
plt.show()
```



## **49.1 References**

Keisling, T.C., 1982. Calculation of the Length of Day 1. Agronomy Journal, 74(4), pp.758-759.

## 50 Solar Radiation

The incident solar radiation on given location is highly influenced by the latitudinal position. In this exercise we examine a method to derive the estimated maximum solar irradiance for each day of the year from a timeseries of observations.

```
# Import modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Load sample data
df = pd.read_csv("../datasets/KS_Manhattan_6_SSW.csv")
df.head()
```

|   | WBANNO | LST_DATE | CRX_VN | LONGITUDE | LATITUDE | T_DAILY_MAX | T_DAILY_M |
|---|--------|----------|--------|-----------|----------|-------------|-----------|
| 0 | 53974  | 20031001 | 1.201  | -96.61    | 39.1     | -9999.0     | -9999.0   |
| 1 | 53974  | 20031002 | 1.201  | -96.61    | 39.1     | 18.9        | 2.5       |
| 2 | 53974  | 20031003 | 1.201  | -96.61    | 39.1     | 22.6        | 8.1       |
| 3 | 53974  | 20031004 | 1.201  | -96.61    | 39.1     | 22.6        | 3.8       |
| 4 | 53974  | 20031005 | 1.201  | -96.61    | 39.1     | 25.0        | 10.6      |

```
# Convert date string to pandas datetime format
df["LST_DATE"] = pd.to_datetime(df["LST_DATE"], format="%Y%m%d")
df["LST_DATE"].head() # Check our conversion.
```

```
0    2003-10-01
1    2003-10-02
2    2003-10-03
3    2003-10-04
4    2003-10-05
Name: LST_DATE, dtype: datetime64[ns]
```

```

# Convert missing values represented as -9999 to NaN
df[df == -9999] = np.nan

# Add year, month, and day of the year to summarize data in future steps.
df["YEAR"] = df["LST_DATE"].dt.year
df["MONTH"] = df["LST_DATE"].dt.month
df["DOY"] = df["LST_DATE"].dt.dayofyear

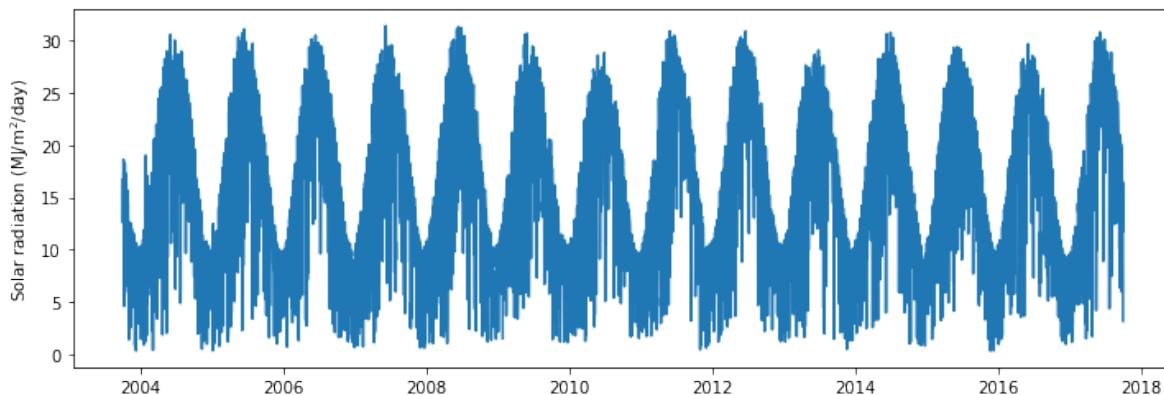
```

## 50.1 Inspect timeseries

```

# Observe trends in solar radiation data
plt.figure(figsize=(12,4))
plt.plot(df["LST_DATE"], df["SOLARAD_DAILY"])
plt.ylabel("Solar radiation (MJ/m$^2$/day) " )
plt.show()

```



## 50.2 Clear sky irradiance (empirical)

We will extract the highest records using a moving filter to estimate the clear sky radiation from observations. We can easily do this with Pandas. Our resulting values will still contain some minor oscillations and the output will vary depending on the size of the window.

```

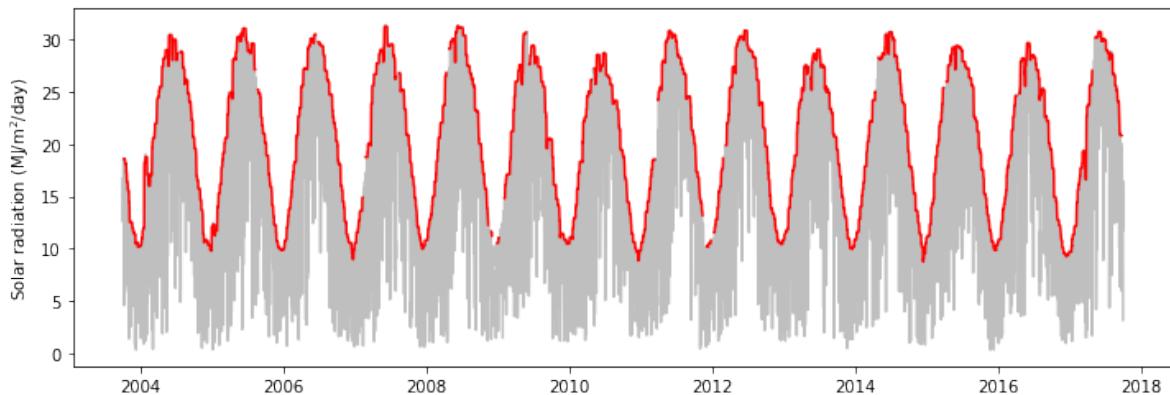
# clear sky solar radiation from observations
df["Rso_obs"] = df["SOLARAD_DAILY"].rolling(window=15, center=True).quantile(0.99)

```

```

# Observe trends in solar radiation data
plt.figure(figsize=(12,4))
plt.plot(df["LST_DATE"], df["SOLARAD_DAILY"], '-k', alpha=0.25)
plt.plot(df["LST_DATE"], df["Rso_obs"], '-r')
plt.ylabel("Solar radiation (MJ/m$^2$/day) " )
plt.show()

```



### 50.3 Clear sky irradiance (from latitude)

Another alternative is to compute the soil radiation based on latitude and elevation. The first step consists of computing the extraterrestrial radiation for daily periods as defined in Eq. 21, FAO-56, and in a subsequent step compute the clear sky solar radiation.

$$Ra = 24(60)/\pi \ Gsc \ dr(\omega \sin(\phi) \sin(\delta) + \cos(\phi) \cos(\delta) \sin(\omega))$$

$Ra$  = extraterrestrial radiation (MJ / m<sup>2</sup> /day)

$Gsc$  = 0.0820 solar constant (MJ/m<sup>2</sup>/min)

$dr = 1 + 0.033 \cos(\frac{2\pi J}{365})$  is the inverse relative distance Earth-Sun

$J$  = day of the year

$\phi$  =  $\pi/180Lat$  latitude in radians

$\delta = 0.409 \sin((2\pi J/365) - 1.39)$  is the solar decimation (rad)

$\omega = \pi/2 - (\arccos(-\tan(\phi) \tan(\delta)))$  is the sunset hour angle (radians)

```

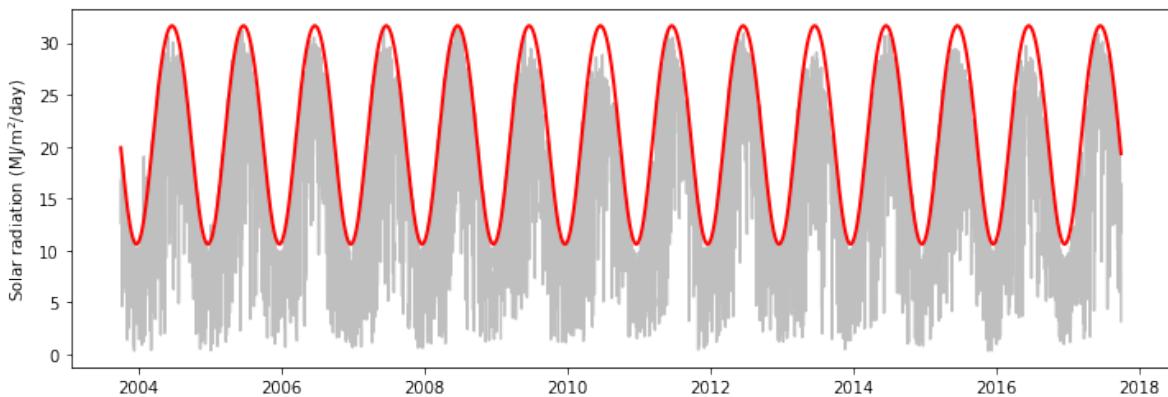
latitude = 39.1949      # Decimal degrees North
elevation = 300          # meters above sea level
J = df["DOY"]

# Step 1: Extraterrestrial solar radiation
phi = np.pi/180 * latitude           # Eq. 22, FAO-56
dr = 1 + 0.033 * np.cos(2*np.pi*J/365) # Eq. 23, FAO-56
d = 0.409*np.sin((2*np.pi * J/365) - 1.39)
omega = (np.arccos(-np.tan(phi)*np.tan(d)))
Gsc = 0.0820
Ra = 24*(60)/np.pi * Gsc * dr * (omega*np.sin(phi)*np.sin(d) + np.cos(phi)*np.cos(d)*np.sin(phi)*np.sin(d))

# Step 2: Clear Sky Radiation: Rso (MJ/m2/day)
df["Rso_lat"] = (0.75 + (2*10**-5)*elevation)*Ra # Eq. 37, FAO-56

# Plot clear sky using latitude and elevation
plt.figure(figsize=(12,4))
plt.plot(df["LST_DATE"], df["SOLARAD_DAILY"], '-k', alpha=0.25)
plt.plot(df["LST_DATE"], df["Rso_lat"], '-r', linewidth=2)
plt.ylabel("Solar radiation (MJ/m$^2$/day) ")
plt.show()

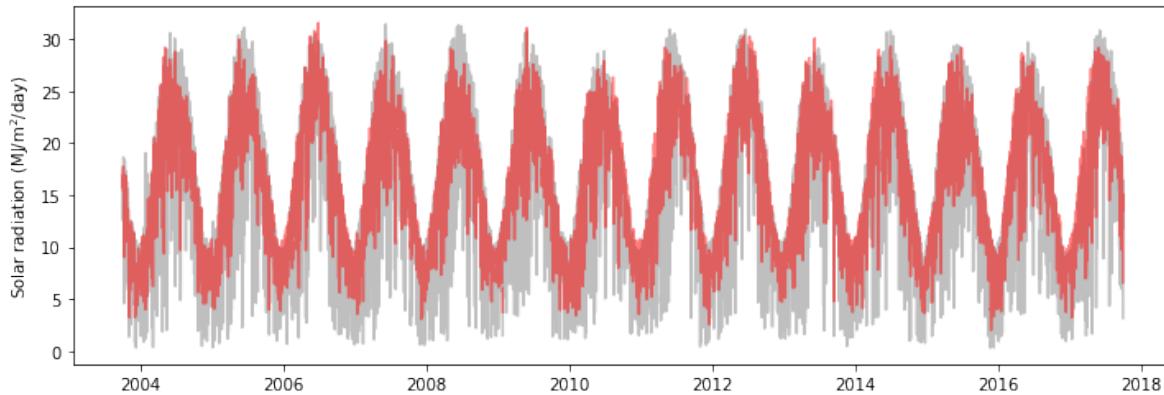
```



## 50.4 Actual solar irradiance (from air temperature)

```
# Clear sky from air temperature observations
df["Rso_temp"] = np.minimum(0.16*Ra*(df["T_DAILY_MAX"]-df["T_DAILY_MIN"]))**0.5, Rso) # Eq.

plt.figure(figsize=(12,4))
plt.plot(df["LST_DATE"], df["SOLARAD_DAILY"], '-k', alpha=0.25)
plt.plot(df["LST_DATE"], df["Rso_temp"], '-r', alpha=0.5)
plt.ylabel("Solar radiation (MJ/m$^2$/day) " )
plt.show()
```



## 50.5 Clear sky solar radiation for each DOY

```
# Compute maximum solar radiation (proxy for ideal, non-cloudy conditions)
Rso_doy = df.groupby("DOY")[["SOLARAD_DAILY","Rso_temp","Rso_obs","Rso_lat"]].mean()
Rso_doy.head()
```

| DOY | SOLARAD_DAILY | Rso_temp | Rso_obs   | Rso_lat   |
|-----|---------------|----------|-----------|-----------|
| 1   | 7.840714      | 7.915730 | 10.522923 | 10.834404 |
| 2   | 7.135000      | 8.016919 | 10.558046 | 10.876334 |
| 3   | 6.400769      | 8.236705 | 10.634077 | 10.921631 |
| 4   | 6.673077      | 6.975503 | 10.738108 | 10.970287 |
| 5   | 7.897692      | 7.595476 | 10.751031 | 11.022290 |

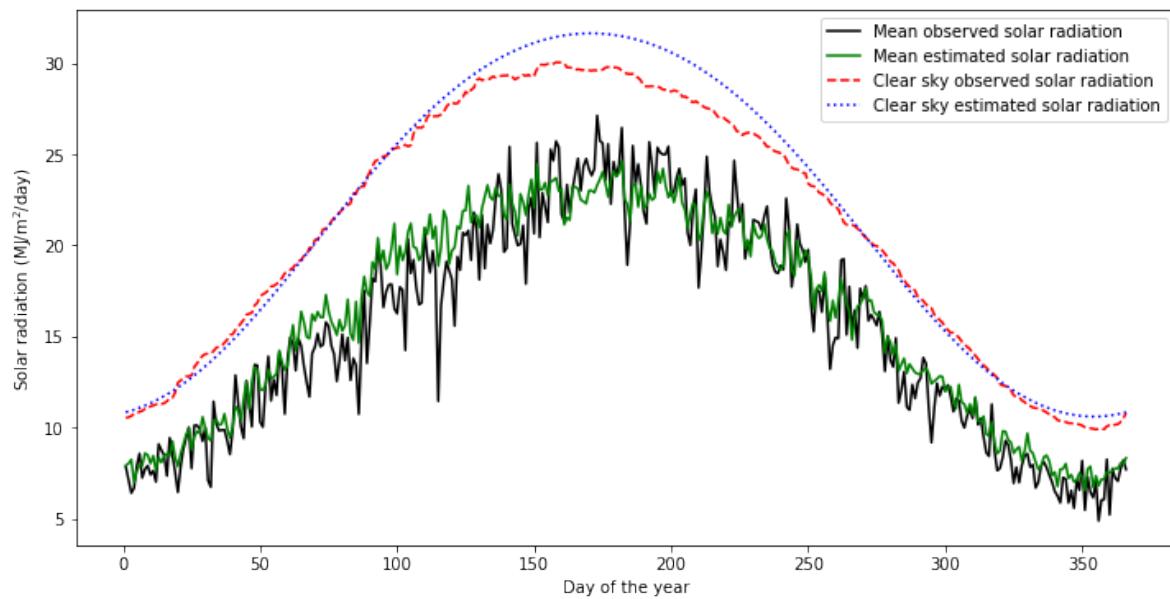
```

plt.figure(figsize=(12,6))
plt.plot(Rso_doy.index, Rso_doy["SOLARAD_DAILY"], '-k', label="Mean observed solar radiation")
plt.plot(Rso_doy.index, Rso_doy["Rso_temp"], '-g', label="Mean estimated solar radiation")

plt.plot(Rso_doy.index, Rso_doy["Rso_obs"], '--r', label="Clear sky observed solar radiation")
plt.plot(Rso_doy.index, Rso_doy["Rso_lat"], ':b', label="Clear sky estimated solar radiation")

plt.xlabel("Day of the year")
plt.ylabel("Solar radiation (MJ/m$^2$/day)")
plt.legend()
plt.show()

```



# 51 Potential Evapotranspiration

Soil evaporation and plant transpiration are two important components of the soil water balance. Evapotranspiration is the sum of these two evaporative losses (one from the soil and the other from the plant stomates). Measuring and modeling soil evaporation and transpiration individually is usually harder than estimating the combined total of these two variables. For instance, if we measure the soil water content in the soil profile yesterday and today, assuming that there was not rainfall and that deep drainage was negligible, we could assume that the change in soil water storage was caused by evapotrasnpiration. In other words, the water loss from the soil was (mainly) by evaporation and plant transpiration.

If we don't need to know the exact partition between evaporation and trasnpiration, then this simple method will provide an estimation of the evapotranspiration between yesterday and today. This would be the **actual evapotranspiration**.

In agriculture, there is another term, the **potential evapotrasnpiration** (PET), which refers to the potential evaporative losses, typically by an reference crop of idealized conditions (e.g. green grass of 8–15 cm height) actively growing under non limiting soil moisture conditions and under the influence of the local weather. This value is used as reference of the maximum water loss from the system due to soil evaporation and plant transpiration.

Knowing the potential evaporation is useful to characterize climate regimes, usually the ratio of annual precipitation to annual PET. Another common use of PET, is in combination with empirical coefficients that can be used to weigh the PET of the reference crop to another crop.

In this notebook we will use daily weather observations to explore several models. Some models, like the Thornthwaite model, work at a monthly scale, but because daily meteorological data is ubiquitous we will only focus on daily computations. Still monthly approaches can be valuable in locations where there are limited or none weather stations.

Over the year scientists have developed multiple ways of estimating PET. A rather convenie

Potetnial evapotrasnpiration is measured using field lysimeters and modeled using meterological information. One of the first models was proposed by John Dalton in 1802, and the most widely used model was proposed by Howard Penman with modifications by John Monteith in the 1950s. In 1998, the Food and Agriculture Organization developed a standard method for computing PET using the Penman-Monteith method.

```
# Import modules
import pandas as pd
import numpy as np
from bokeh.plotting import figure,show,output_notebook
output_notebook()
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_1

Because potential evapotranspiration represents potential evaporative losses, the atmospheric vapor pressure deficit is a common term in many models. The vapor pressure deficit is computed by subtracting the actual vapor pressure from the saturation vapor pressure at the same ambient temperature. Vapor pressure deficit can be easily approximated using air temperature and relative humidity using Tetens equation for the saturation vapor pressure.

In order to use the Tetens equation in multiple functions, we will first define the Tetens function separately. You can include the Tetens equation as part of the PET function, in order to create a single, stand-alone function for your projects. here this approach is more convenient since we are comparing multiple models.

```
# Auxiliary functions
tetens = lambda T: 0.6108*np.exp(17.27*T/(T+237.3))

# Plot tetens function to illustrate the dependence of saturation vapor to temperature
air_temperature_range = np.arange(50)
f = figure(width=400, height=300)
f.line(air_temperature_range, tetens(air_temperature_range))
f.xaxis.axis_label = 'Air Temperature (Celsius)'
f.yaxis.axis_label = 'Saturation Vapor Pressure (kPa)'
show(f)
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_1

## 51.1 Dalton model

In 1802 John Dalton proposed a simple model for predicting potential evaporation based on wind speed and vapor pressure deficit. This equation works well on open water bodies, but it does not account for the effect of plants and the presence of soils. Several equations have been based on the formulation proposed by Dalton, in which additional coefficients and exponents aim at better predicting local conditions.

Dalton started as a meteorologist, frequently recording atmospheric conditions in his own journals. The equation proposed by Dalton is considered a mass transfer model, the name emphasizing that water vapor moves along a vapor gradient and that this gradient can be maintained or enhanced by wind replacing the saturated air near the evaporating surface with new air volume containing lower water vapor.

$$E = u(e_s - e_a)$$

$u$  is the wind speed in m/s

$e_s$  is the atmospheric saturation vapor pressure in kPa

$e_a$  is the actual atmospheric vapor pressure

```
## John Dalton (1802)

def dalton(T_min,T_max,RH_min,RH_max,wind_speed):
    e_sat = (tetens(T_min) + tetens(T_max)) / 2
    e_atm = (tetens(T_min)*(RH_max/100) + tetens(T_max)*(RH_min/100)) / 2
    PE = (3.648 + 0.7223*wind_speed)*(e_sat - e_atm)
    return PE
```

## 51.2 Romanenko model

```
## Romanenko (1961)

def romanenko(T_min,T_max,RH_min,RH_max):
    T_avg = (T_min + T_max)/2
    RH_avg = (RH_min + RH_max)/2
    PET = 0.00006*(25 + T_avg)**2*(100 - RH_avg)
    return PET
```

### 51.3 Penman model

```
## Penman (1948)

def penman(T_min,T_max,RH_min,RH_max,wind_speed):
    e_sat = (tetens(T_min) + tetens(T_max)) / 2
    e_atm = (tetens(T_min)*(RH_max/100) + tetens(T_max)*(RH_min/100)) / 2
    PET = (2.625 + 0.000479/wind_speed)*(e_sat - e_atm)
    return PET
```

### 51.4 Jensen model

```
## Jensen-Haise (1963)

def jensen_haise(T_min,T_max,solar_rad):
    T_avg = (T_min + T_max)/2
    PET = 0.0102*(T_avg+3)*solar_rad
    return PET
```

### 51.5 Hargreaves model

$$PET = 0.0023 R_a (T_{avg} + 17.8) \sqrt{(T_{max} - T_{min})}$$

$R_a$  is the extraterrestrial solar radiation ( $MJ/m^2$ )

$T_{max}$  is the maximum daily air temperature

$T_{min}$  is the minimum daily air temperature

$T_{avg}$  is the average daily air temperature

```
## Hargreaves (1982)

def hargreaves(T_min,T_max,doy,latitude):

    # Computation of extra-terrestrial solar radiation
    dr = 1 + 0.033 * np.cos(2 * np.pi * doy/365) # Inverse relative distance Earth-Sun
    phi = np.pi / 180 * latitude # Latitude in radians
    d = 0.409 * np.sin((2 * np.pi * doy/365) - 1.39) # Solar declination
```

```

omega = np.arccos(-np.tan(phi) * np.tan(d)) # Sunset hour angle
Gsc = 0.0820 # Solar constant
Ra = 24 * 60 / np.pi * Gsc * dr * (omega * np.sin(phi) * np.sin(d) + np.cos(phi) * np.
T_avg = (T_min + T_max)/2
PET = 0.0023 * Ra * (T_avg + 17.8) * (T_max - T_min)**0.5
return PET

```

### 51.5.1 Priestley-Taylor model

This model was developed with the intention of reducing the number of variables required to compute the potential evapotranspiration in regions with limited environmental observations as an alternative to the model proposed by Penman-Monteith. Thus, several terms of the PM equation are condensed into an empirical constant. The psychrometric constant is computed as:

$$\gamma = \frac{C_p P}{\epsilon \lambda} \approx 6.6510^{-4} P$$

$\gamma$  = is the psychrometric constant in kPa/°C

$C_p$  is the specific heat at constant pressure 0.001013 (MJ kg<sup>-1</sup> °C<sup>-1</sup>)

$\epsilon$  is the ratio molecular weight of water vapour/dry air = 0.622.

$\lambda$  is the latent heat of vaporization, 2.45 MJ kg<sup>-1</sup>

$P$  is the atmospheric pressure in kPa either computed from the station altitude or measured by a barometer.

The term  $(R_n - G)$  represents the available energy

```

## Priestley-Taylor (1972)

def priestley_taylor(T_min,T_max,solar_rad,altitude):
    T_avg = (T_min + T_max)/2
    atm_pressure = 101.3 * ((293 - 0.0065 * altitude)/293)**5.26 # kPa
    gamma = 0.000665 * atm_pressure # Psychrometric constant Cp/(2.45 * 0.622) = 0.000665
    delta = 4098 * (0.6108 * np.exp(17.27 * T_avg / (T_avg + 237.3))) / (T_avg + 237.3)*
    soil_heat_flux = 0;
    alpha = 0.5
    PET = alpha*delta/(delta+gamma)*(solar_rad-soil_heat_flux)
    return PET

```

## 51.6 Penman-Monteith model

```
def penman_monteith(T_min,T_max,RH_min,RH_max,solar_rad,wind_speed,doy,latitude,altitude):
    T_avg = (T_min + T_max)/2
    atm_pressure = 101.3 * ((293 - 0.0065 * altitude) / 293)**5.26 # Can be also obtained
    Cp = 0.001013; # Approx. 0.001013 for average atmospheric conditions
    epsilon = 0.622
    Lambda = 2.45
    gamma = (Cp * atm_pressure) / (epsilon * Lambda) # Approx. 0.000665

    ##### Wind speed
    wind_height = 1.5 # Most common height in meters
    wind_speed_2m = wind_speed * (4.87 / np.log((67.8 * wind_height) - 5.42)) # Eq. 47, FAO-56

    ##### Air humidity and vapor pressure
    delta = 4098 * (0.6108 * np.exp(17.27 * T_avg / (T_avg + 237.3))) / (T_avg + 237.3)*
    e_temp_max = 0.6108 * np.exp(17.27 * T_max / (T_max + 237.3)) # Eq. 11, //FAO-56
    e_temp_min = 0.6108 * np.exp(17.27 * T_min / (T_min + 237.3))
    e_saturation = (e_temp_max + e_temp_min) / 2
    e_actual = (e_temp_min * (RH_max / 100) + e_temp_max * (RH_min / 100)) / 2

    ##### Solar radiation
    dr = 1 + 0.033 * np.cos(2 * np.pi * doy/365) # Eq. 23, FAO-56
    phi = np.pi / 180 * latitude # Eq. 22, FAO-56
    d = 0.409 * np.sin((2 * np.pi * doy/365) - 1.39)
    omega = np.arccos(-np.tan(phi) * np.tan(d))
    Gsc = 0.0820 # Approx. 0.0820
    Ra = 24 * 60 / np.pi * Gsc * dr * (omega * np.sin(phi) * np.sin(d) + np.cos(phi) * np.cos(phi))

    # Clear Sky Radiation: Rso (MJ/m2/day)
    Rso = (0.75 + (2 * 10**-5) * altitude) * Ra # Eq. 37, FAO-56

    # Rs/Rso = relative shortwave radiation (limited to <= 1.0)
    alpha = 0.23 # 0.23 for hypothetical grass reference crop
    Rns = (1 - alpha) * solar_rad # Eq. 38, FAO-56
    sigma = 4.903 * 10**-9
    maxTempK = T_max + 273.16
    minTempK = T_min + 273.16
    Rnl = sigma * (maxTempK**4 + minTempK**4) / 2 * (0.34 - 0.14 * np.sqrt(e_actual)) *
    Rn = Rns - Rnl # Eq. 40, FAO-56
```

```

# Soil heat flux density
soil_heat_flux = 0 # Eq. 42, FAO-56 G = 0 for daily time steps [MJ/m2/day]

# ETo calculation
PET = (0.408 * delta * (solar_rad - soil_heat_flux) + gamma * (900 / (T_avg + 273)) *
return np.round(PET,2)

```

## 51.7 Data

In this section we will use real data to test the different PET models.

```

# Import data
df = pd.read_csv('../datasets/acme_ok_daily.csv')
df['Date'] = pd.to_datetime(df['Date'], format='%m/%d/%y %H:%M')
df.head()

```

|   | Date       | DOY | TMAX      | TMIN      | RAIN  | HMAX | HMIN  | ATOT | W2AVG    | ETgrass  |
|---|------------|-----|-----------|-----------|-------|------|-------|------|----------|----------|
| 0 | 2005-01-01 | 1   | 21.161111 | 14.272222 | 0.00  | 97.5 | 65.97 | 4.09 | 5.194592 | 1.976940 |
| 1 | 2005-01-02 | 2   | 21.261111 | 4.794444  | 0.00  | 99.3 | 77.37 | 4.11 | 3.428788 | 1.302427 |
| 2 | 2005-01-03 | 3   | 5.855556  | 3.477778  | 2.54  | 99.8 | 98.20 | 2.98 | 3.249973 | 0.349413 |
| 3 | 2005-01-04 | 4   | 4.644444  | 0.883333  | 7.62  | 99.6 | 98.50 | 1.21 | 3.527137 | 0.288802 |
| 4 | 2005-01-05 | 5   | 0.827778  | -9.172222 | 24.13 | 99.4 | 86.80 | 1.65 | NaN      | 0.367956 |

```

dalton_pred = dalton(df['TMIN'], df['TMAX'], df['HMIN'], df['HMAX'], df['W2AVG'])
penman_pred = penman(df['TMIN'], df['TMAX'], df['HMIN'], df['HMAX'], df['W2AVG'])

# Plot models
f = figure(width=600, height=400, x_axis_type="datetime")
f.line(df['Date'], dalton_pred, legend_label='Dalton', line_color='tomato')
f.line(df['Date'], penman_pred, legend_label='Penman')
show(f)

```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_e

## 51.8 Practice

- Compute the potential evaporation or potential evapotranspiration with the other models. Which models do you think produce the most reasonable results for ET predictions over terrestrial ecosystems?
- Using the Penman-Monteith model, what is the impact of wind speed? By how many millimeters per day would the potential ET increased when incrementing the wind speed by 1 m/s?

## 51.9 References

Dalton J (1802) Experimental essays on the constitution of mixed gases; on the force of steam of vapour from waters and other liquids in different temperatures, both in a Torricellian vacuum and in air on evaporation and on the expansion of gases by heat. Mem Manch Lit Philos Soc 5:535–602

Hargreaves G (1989) Preciseness of estimated potential evapotranspiration. J Irrig Drain Eng 115(6):1000–1007

Penman HC (1948) Natural evaporation from open water, bare soil and grass. Proc R Soc Lond Ser A 193:120–145

Thornthwaite, C.W., 1948. An approach toward a rational classification of climate. Geographical review, 38(1), pp.55-94.

McMahon, T.A., Finlayson, B.L. and Peel, M.C., 2016. Historical developments of models for estimating evaporation using standard meteorological data. Wiley Interdisciplinary Reviews: Water, 3(6), pp.788-818.

# 52 Soil Temperature Model

Soil temperature is one of those variables that follow the theory closely. In this exercise we will implement an analytical solution of the heat conduction-diffusion equation. The model is simple and accounts for soil temperature in time and soil depth. This model is robust enough to be used for research.

The model states that the annual mean temperature at all soil depths is the same. At increasing depths the thermal amplitude decreases and the timing of maximum and minimum temperatures experience a delay. This makes sense since it takes time for heat to move from the surface to deeper layers and the thermal fluctuations are higher near the surface than at depth as a consequence of heat losses primarily due to radiation and convection.

## 52.1 Model

$$T(z, t) = T_{avg} + A e^{-z/d} \sin(\omega t - z/d - \phi)$$

$T$  is the soil temperature at time  $t$  and depth  $z$

$z$  is the soil depth in meters

$t$  is the time in days of year

$T_{avg}$  is the annual average temperature at the soil surface

$A$  is the thermal amplitude:  $(T_{max} + T_{min})/2$ .

$\omega$  is the angular frequency:  $2\pi/P$

$P$  is the period. Should be in the same units as  $t$ . The period is 365 days for annual oscillations and 24 hours for daily oscillations.

$\phi$  is the phase constant, which is defined as:  $\frac{\pi}{2} + \omega t_0$

$t_0$  is the time lag from an arbitrary starting point. In this case are days from January 1.

$d$  is the damping depth, which is defined as:  $\sqrt{(2D/\omega)}$ . It has length units.

$D$  is thermal diffusivity in  $m^2 d^{-1}$ . The thermal diffusivity is defined as  $\kappa / C$

$\kappa$  is the soil thermal conductivity in  $J m^{-1} K^{-1} d^{-1}$

$C$  is the soil volumetric heat capacity in  $Jm^{-3}K^{-1}$

## 52.2 Assumptions

- Constant soil thermal diffusivity.
- Uniform soil texture
- Temperature in deep layers approximate the mean annual air temperature
- In situation where we don't have observations of soil temperature at the surface we also assume that the soil surface temperature is equal to the air temperature.

```
# Import modules
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

## 52.3 Model inputs

```
# Constants
T_avg = 25 # Annual average temperature at the soil surface
A0 = 10 # Annual thermal amplitude at the soil surface
D = 0.203 # Thermal diffusivity obtained from KD2 Pro instrument [mm^2/s]
D = D / 100 * 86400 # convert to cm^2/day
period = 365 # days
omega = 2*np.pi/period
t_0 = 15 # Time lag in days from January 1
phi = np.pi/2 + omega*t_0 # Phase constant
d = (2*D/omega)**(1/2) # Damping depth
D
```

175.39200000000002

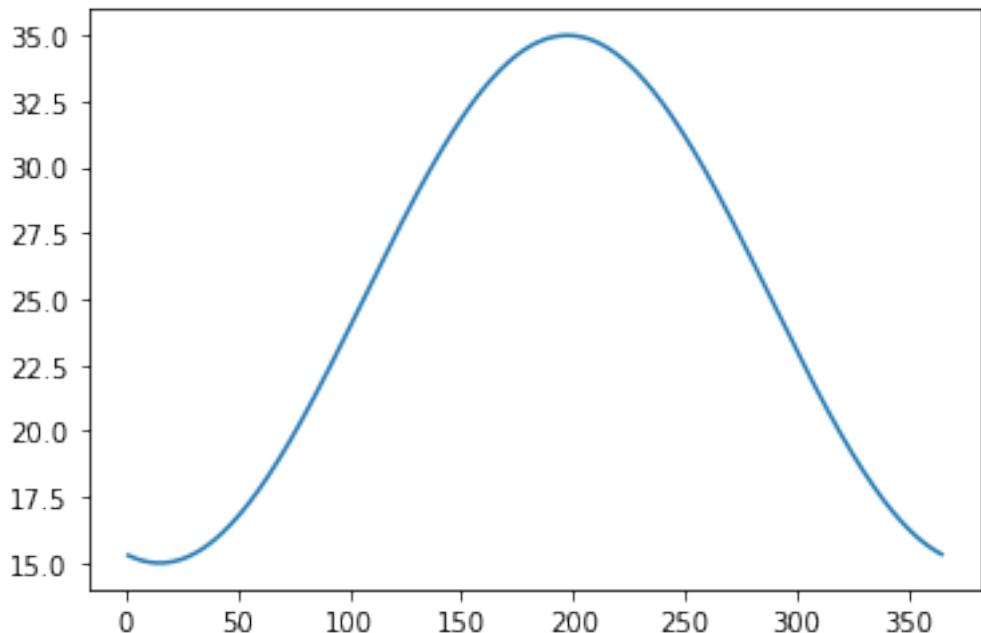
## 52.4 Define model

```
# Define model as lambda function
T_soilfn = lambda doy,z: T_avg + A0 * np.exp(-z/d) * np.sin(omega*doy - z/d - phi)
```

## 52.5 Soil temperature for a specific depth as a function of time

```
doy = np.arange(1,366)
z = 0
T_soil = T_soilfn(doy,z)

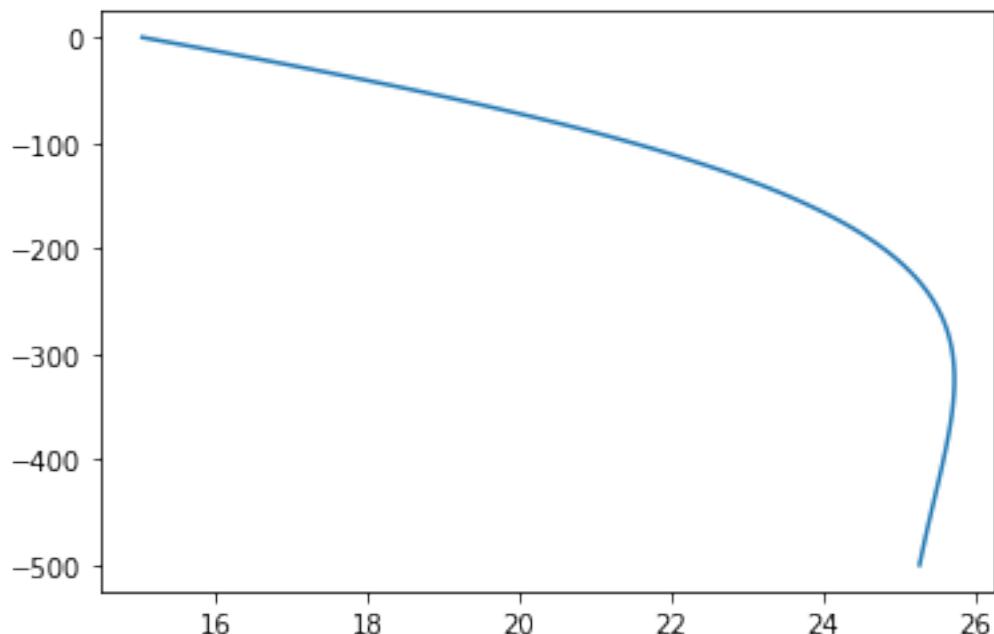
# Plot
plt.figure()
plt.plot(doy,T_soil)
plt.show()
```



## 52.6 Soil temperature for a specific day of the year as a function of depth

```
doy = 10
Nz = 100 # Number of interpolation
zmax = 500 # cm
z = np.linspace(0,zmax,Nz)
T = T_soilfn(doy,z)

plt.figure()
plt.plot(T,-z)
plt.show()
```



## 52.7 Soil temperature as a function of both DOY and depth

```
doy = np.arange(1,366)
z = np.linspace(0,500,1000)
doy_grid,z_grid = np.meshgrid(doy,z)
```

```

# Predict soil temperature for each grid
T_grid = T_soilfn(doy_grid,z_grid)

# Create figure
fig = plt.figure(figsize=(10, 6), dpi=80) # 10 inch by 6 inch dpi = dots per inch

# Get figure axes and convert it to a 3D projection
ax = fig.gca(projection='3d')

# Add surface plot to axes. Save this surface plot in a variable
surf = ax.plot_surface(doy_grid, z_grid, T_grid, cmap='viridis', antialiased=False)

# Add colorbar to figure based on ranges in the surf map.
fig.colorbar(surf, shrink=0.5, aspect=20)

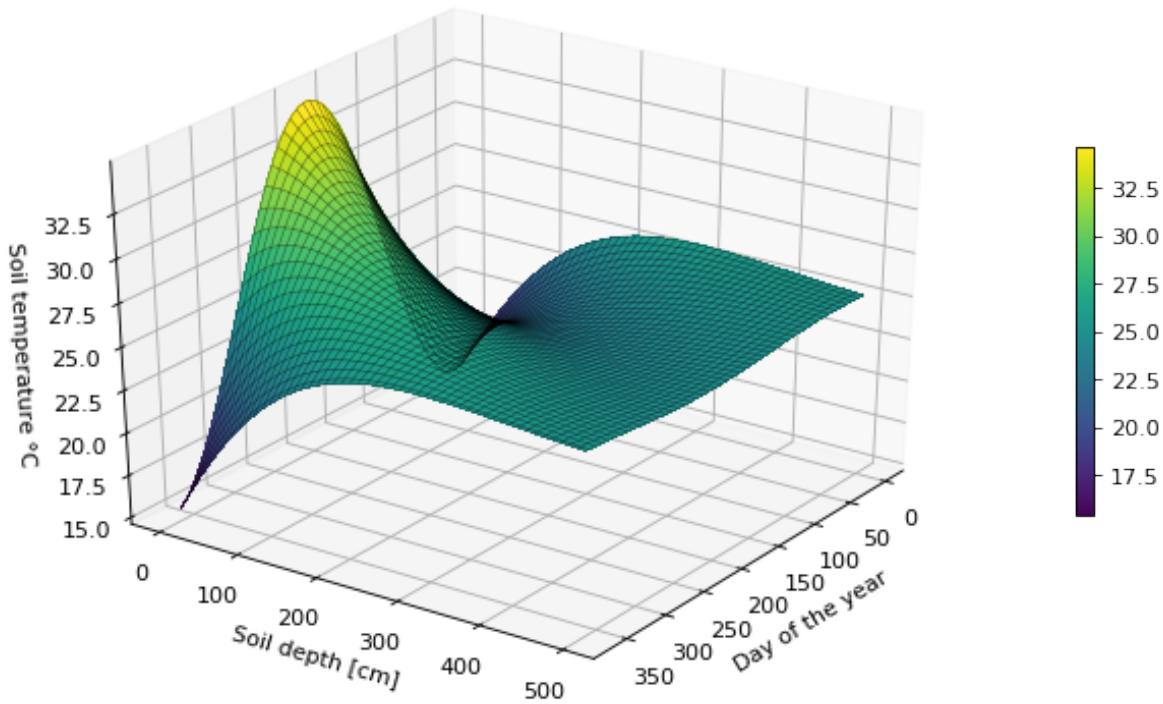
# Wire mesh
frame = surf = ax.plot_wireframe(doy_grid, z_grid, T_grid, linewidth=0.5, color='k', alpha=0.5)

# Label x,y, and z axis
ax.set_xlabel("Day of the year")
ax.set_ylabel('Soil depth [cm]')
ax.set_zlabel('Soil temperature \N{DEGREE SIGN}C')

# Set position of the 3D plot
ax.view_init(elev=30, azim=35) # elevation and azimuth. Change their value to see what happens

plt.show()

```



## 52.8 Interactive plots

```
from bokeh.plotting import figure, show, output_notebook, ColumnDataSource, save
from bokeh.layouts import row
from bokeh.models import HoverTool
from bokeh.io import export_svgs
output_notebook()
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_l

```
# Set data for p1
doy = np.arange(1,366)
z = 0
source_p1 = ColumnDataSource(data=dict(x=doy, y=T_soilfn(doy,z)))
```

```

# Define tools for p1
hover_p1 = HoverTool(
    tooltips=[
        ("Time (days)", "@x{0.}"),
        ("Temperature (Celsius)", "@y{0.00}"))
)
)

# Create plots
p1 = figure(y_range=[0,50],
            width=400,
            height=300,
            title="Soil Temperature as a Function of Time",
            tools=[hover_p1],
            toolbar_location="right")

p1.xaxis.axis_label = 'Time [hours]'
p1.yaxis.axis_label = 'Temperature'
p1.line('x','y',source=source_p1)

# Set data for p2
doy = 150
z = np.linspace(0,500,100)
source_p2 = ColumnDataSource(data=dict(y=-1*z, x=T_soilfn(150,z)))

# Define tools for p1
hover_p2 = HoverTool(
    tooltips=[
        ("Depth (cm)", "@y{0.0}"),
        ("Temperature (Celsius)", "@x{0.00}"))
)
)

# Create plots
p2 = figure(y_range=[0,-500],
            width=400,
            height=300,
            title="Soil Temperature as a Function of Soil Depth",
            tools=[hover_p2],
            toolbar_location="right")

```

```
p2.xaxis.axis_label = 'Temperature'  
p2.yaxis.axis_label = 'Depth (cm)'  
p2.min_border_left = 100  
p2.line('x','y',source=source_p2)  
  
p1.output_backend = "svg"  
p2.output_backend = "svg"  
  
show(row(p1,p2))
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs\_e

## 52.9 References

Wu, J. and Nofziger, D.L., 1999. Incorporating temperature effects on pesticide degradation into a management model. Journal of Environmental Quality, 28(1), pp.92-100.

## 53 Soil Water Retention Curve

A soil water retention curve represents the relationship between the amount of water in the soil expressed as the volumetric water content and the energy state of the soil water expressed in  $J/kg$ ,  $J/m^3$ , or  $kPa$ .

Soil water retention curves are determined empirically by collecting soil samples and using a series of instruments to quantify the soil water content at different energy levels.

The most popular model is that proposed by van Genuchten in 1980:

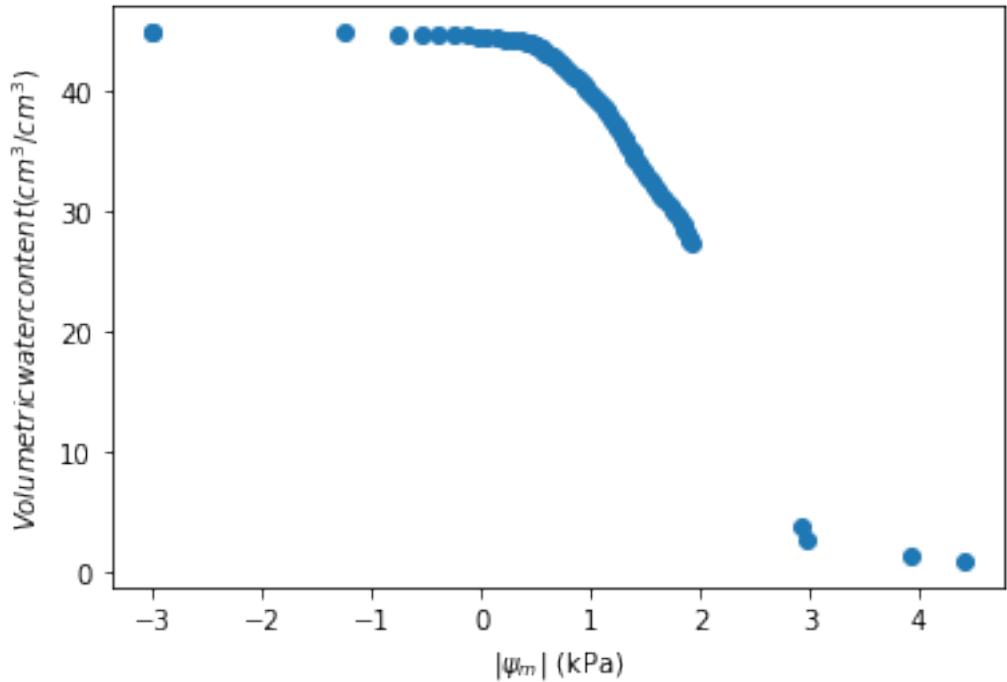
$$\frac{\theta - \theta_r}{\theta_s - \theta_r} = [1 + (-\alpha\psi_m)^n]^{-m}$$

```
# Import modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

df = pd.read_csv('../datasets/soil_water_retention_curve.csv')
df.head()
```

|   | matric | theta |
|---|--------|-------|
| 0 | 0.001  | 44.8  |
| 1 | 0.001  | 44.8  |
| 2 | 0.056  | 44.8  |
| 3 | 0.173  | 44.7  |
| 4 | 0.286  | 44.7  |

```
# Plot observations
plt.scatter(np.log10(df["matric"]), df["theta"])
plt.xlabel('$|\psi_m|$ (kPa)')
plt.ylabel("$Volumetric water content (cm^3/cm^3)$")
plt.show()
```



### 53.1 Define soil water retention model

```

model = lambda x, alpha, n, m, theta_r, theta_s: theta_r + (theta_s-theta_r)*(1+(alpha*x))**n

# Test function for a single value of matric potential
test_matric = 150
model(test_matric, 0.002, 1.5, 1, 0.05, 0.5)

```

0.4364927592361794

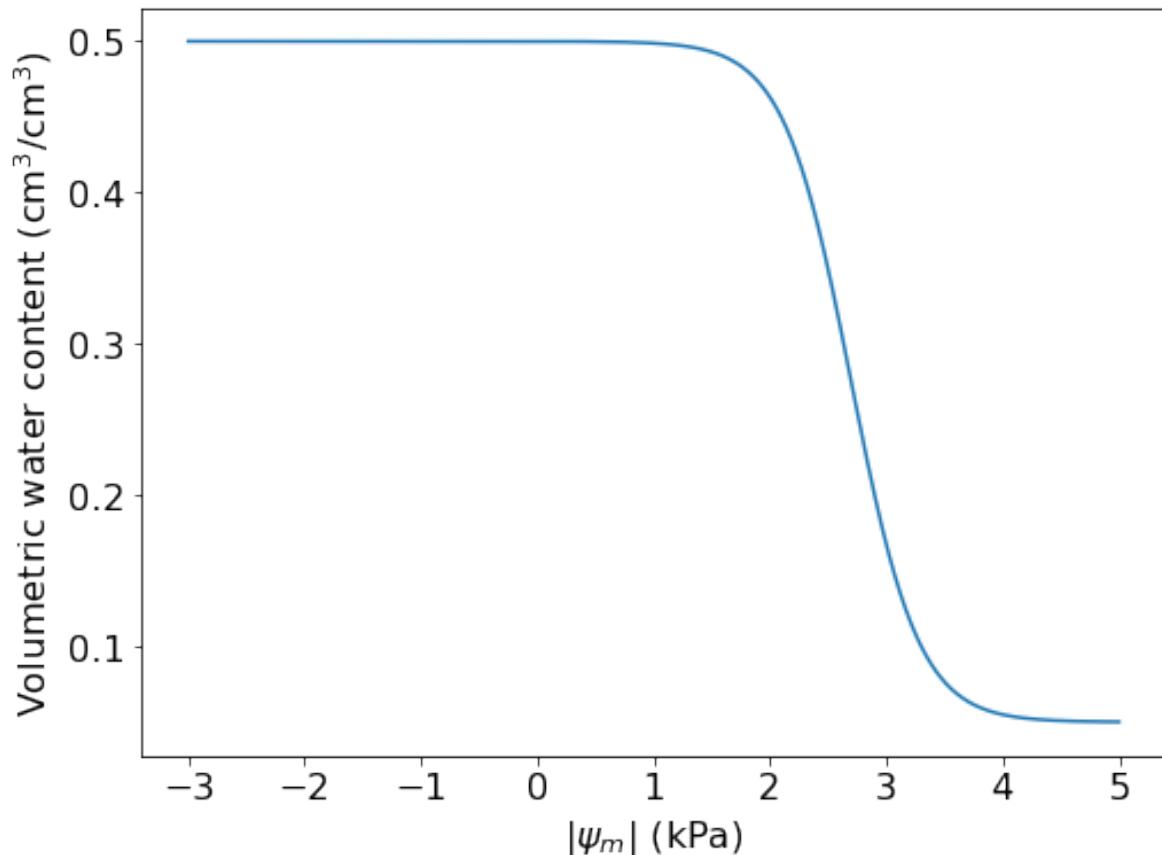
```

# Test function for a range of matric potentials

matric = np.logspace(-3,5,1000)
plt.figure(figsize=(8,6))
plt.plot(np.log10(matric), model(matric,0.002,1.5,1,0.05,0.5))
plt.xlabel('$|\psi_m|$ (kPa)', size=16)
plt.ylabel('Volumetric water content (cm$^3$/cm$^3$)', size=16)
plt.xticks(fontsize=16)

```

```
plt.yticks(fontsize=16)
plt.show()
```

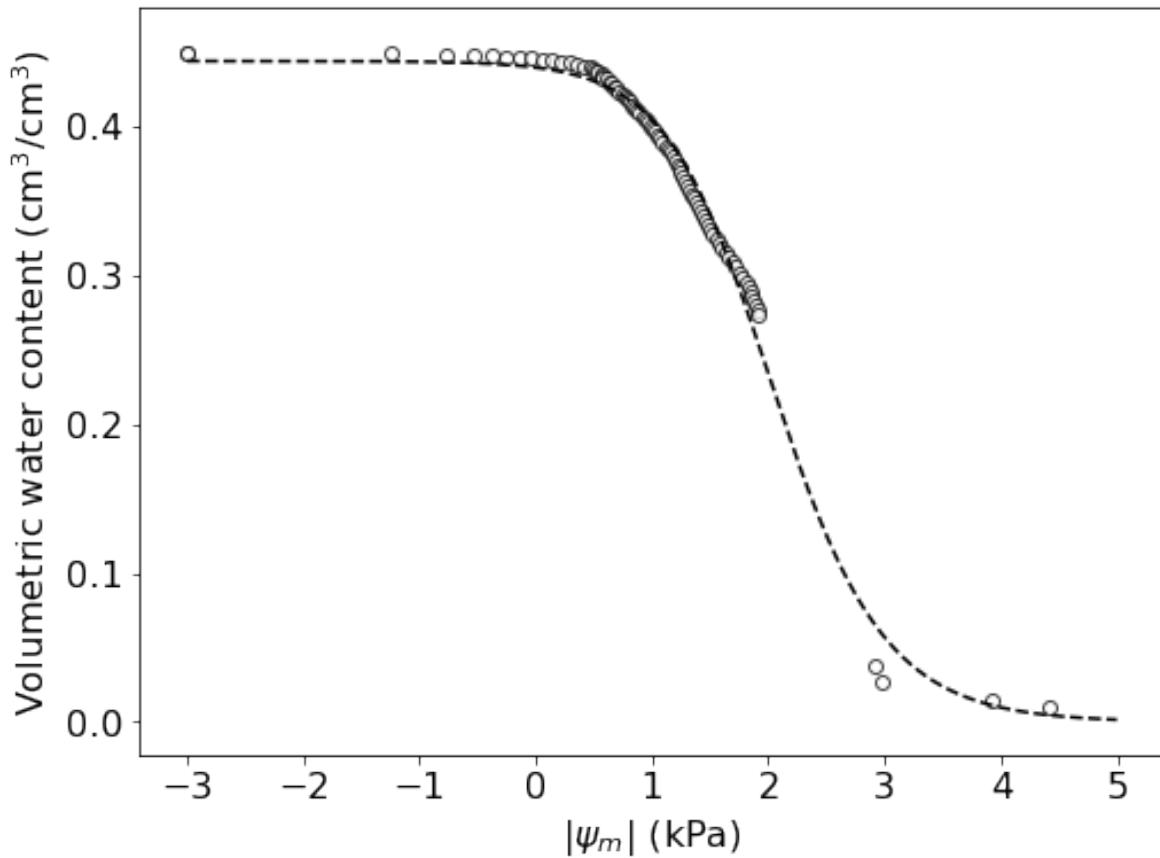


## 53.2 Fit model

```
xdata = df["matric"]
ydata = df["theta"]/100
#p0 = [0.002, 1.5, 1, 0.05, 0.5]
lb = [0, 1, 0, 0, 0.35]
ub = [1, 5, 2, 0.25, 0.55]
bounds=(lb,ub)
par_opt, par_cov = curve_fit(model, xdata, ydata, bounds=bounds)
print(par_opt)
```

```
[1.25680016e-02 1.00000000e+00 7.86928044e-01 5.86306870e-14
4.43268752e-01]
```

```
# Plot results
plt.figure(figsize=(8,6))
plt.scatter(np.log10(xdata), ydata, marker='o', facecolor='w', alpha=0.75, edgecolor='k')
plt.plot(np.log10(matric), model(matric, *par_opt), '--k')
plt.xlabel('|\psi_m| (kPa)', size=16)
plt.ylabel('Volumetric water content (cm^3/cm^3)', size=16)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.show()
```



### **53.3 References**

van Genuchten, M.T., 1980. A closed form equation for predicting hydraulic conductivity of unsaturated soils: Journal of the Soil Science Society of America.

Groenevelt, P.H. and Grant, C.D., 2004. A new model for the soil-water retention curve that solves the problem of residual water contents. European Journal of Soil Science, 55(3), pp.479-485.

## 54 Proctor test

The proctor test is a standardized test that measures the susceptibility to soil compaction. The objective of the test is to identify the maximum dry bulk density value of the soil. Different samples of loose soil are compacted at different moisture contents, but the bulk density is always computed based on dry soil (i.e. after oven-drying the compacted samples).

When the soil is dry the friction between the solid particles prevents achieving high levels of compaction. Increasing amounts of water act as a lubricant and then it is possible compact the soil to a greater degree. Now the particles can move and re-organize to achieve a greater density. Adding excess of water results in lower compaction than at moderate soil moisture levels. Water is an incompressible fluid and thus it absorbs part of the stress, preventing compression. When all the soil pores are filled with water “there is no place to go” for the particles and since water is incompressible, compaction cannot be achieved.

Excessive soil compaction is undesirable in agricultural systems since it diminishes root exploration, reduces soil water storage capacity, creates dense layers that restrict soil water flow, promotes soil crusting, and creates poor seedbed conditions. On the other hand, maximum soil compaction is a desirable property in construction to ensure that building foundations and roads are stable to carry the expected load.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

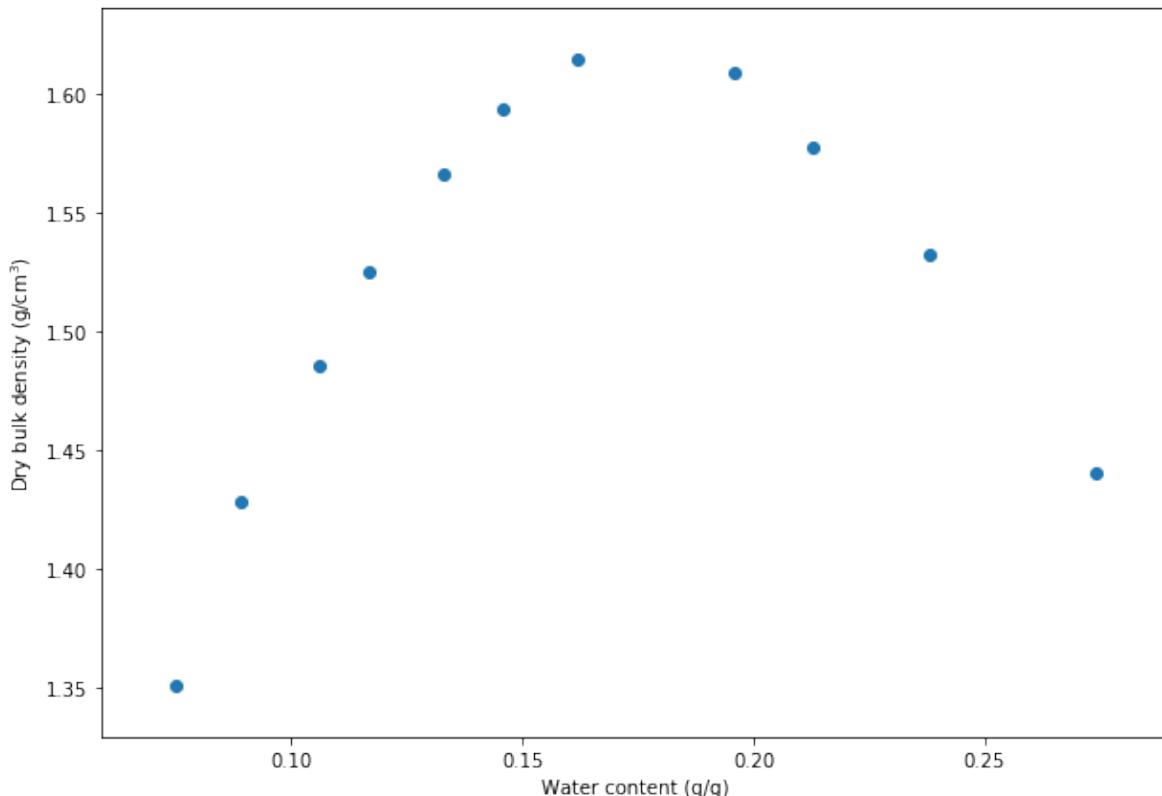
data = pd.read_csv('../datasets/proctor_test.csv', skiprows=[0])
data.head()
```

|   | gravimetric_water | dry_bulk_density |
|---|-------------------|------------------|
| 0 | 0.075             | 1.351            |
| 1 | 0.089             | 1.428            |
| 2 | 0.106             | 1.485            |
| 3 | 0.117             | 1.525            |
| 4 | 0.133             | 1.566            |

```

plt.figure(figsize=(10,7))
plt.scatter(data.gravimetric_water, data.dry_bulk_density)
plt.xlabel('Water content (g/g)')
plt.ylabel('Dry bulk density (g/cm$^3$)')
plt.show()

```



```

par = np.polyfit(data.gravimetric_water, data.dry_bulk_density, 2)
print(par)

```

```
[ -22.08573584   8.03425314   0.88349184]
```

```

polyfun = np.poly1d(par) # Create object
print(polyfun) # polynomial function

```

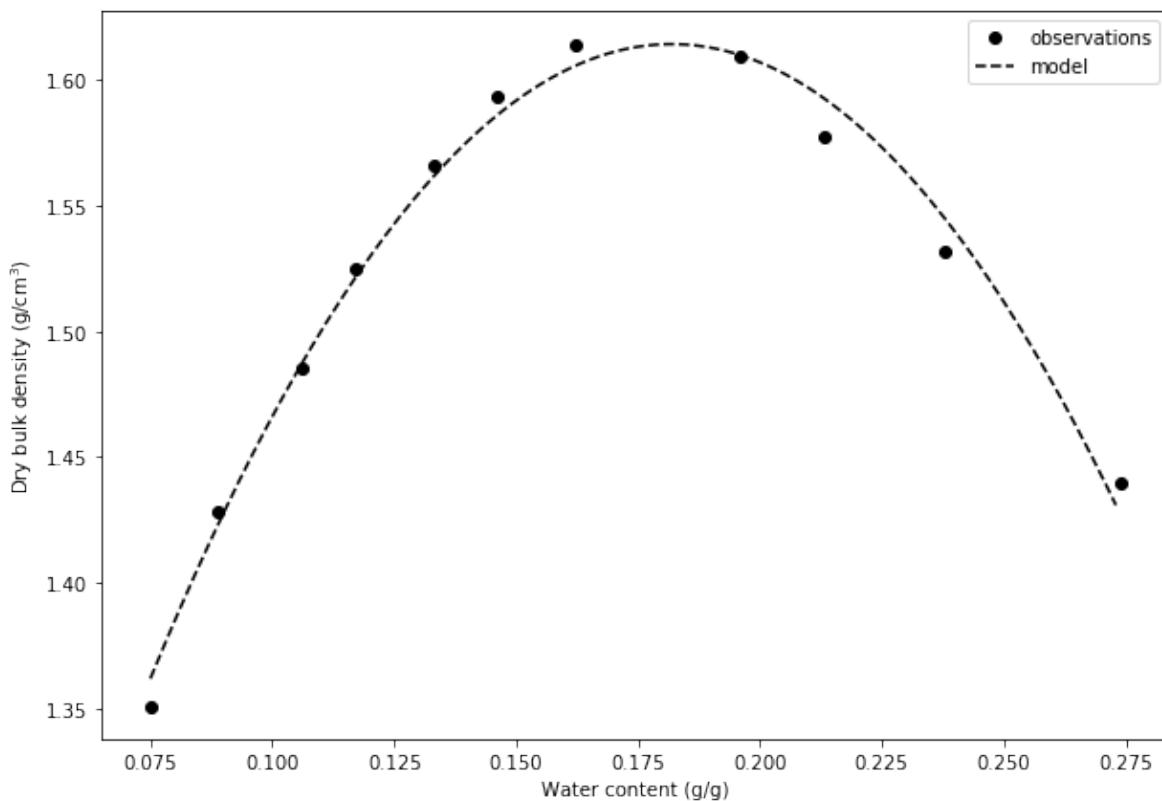
$$\begin{aligned} & 2 \\ & -22.09 x^2 + 8.034 x + 0.8835 \end{aligned}$$

```

# Plot
water_min = np.min(data.gravimetric_water)
water_max = np.max(data.gravimetric_water)
water_pred = np.arange(water_min, water_max, 0.001)
dry_bulk_density_pred = polyfun(water_pred)

plt.figure(figsize=(10,7))
plt.plot(data.gravimetric_water, data.dry_bulk_density, 'ok', label='observations')
plt.plot(water_pred, dry_bulk_density_pred, '--k', label='model')
plt.xlabel('Water content (g/g)')
plt.ylabel('Dry bulk density (g/cm$^3$)')
plt.legend()
plt.show()

```



```

# Find optimal soil moisture and maximum density

first_derivative = np.diff(dry_bulk_density_pred, n=1) # One value shorter

```

```

idx_lowest_derivative = np.argmin(np.abs(first_derivative))

max_density = dry_bulk_density_pred[idx_lowest_derivative]
optimal_water = water_pred[idx_lowest_derivative]

print('Maximum dry bulk density:', round(max_density,2),'g/cm^3')
print('Optimal gravimetric soil water content:', round(optimal_water,3),'g/g')

```

Maximum dry bulk density: 1.61 g/cm<sup>3</sup>  
Optimal gravimetric soil water content: 0.181 g/g

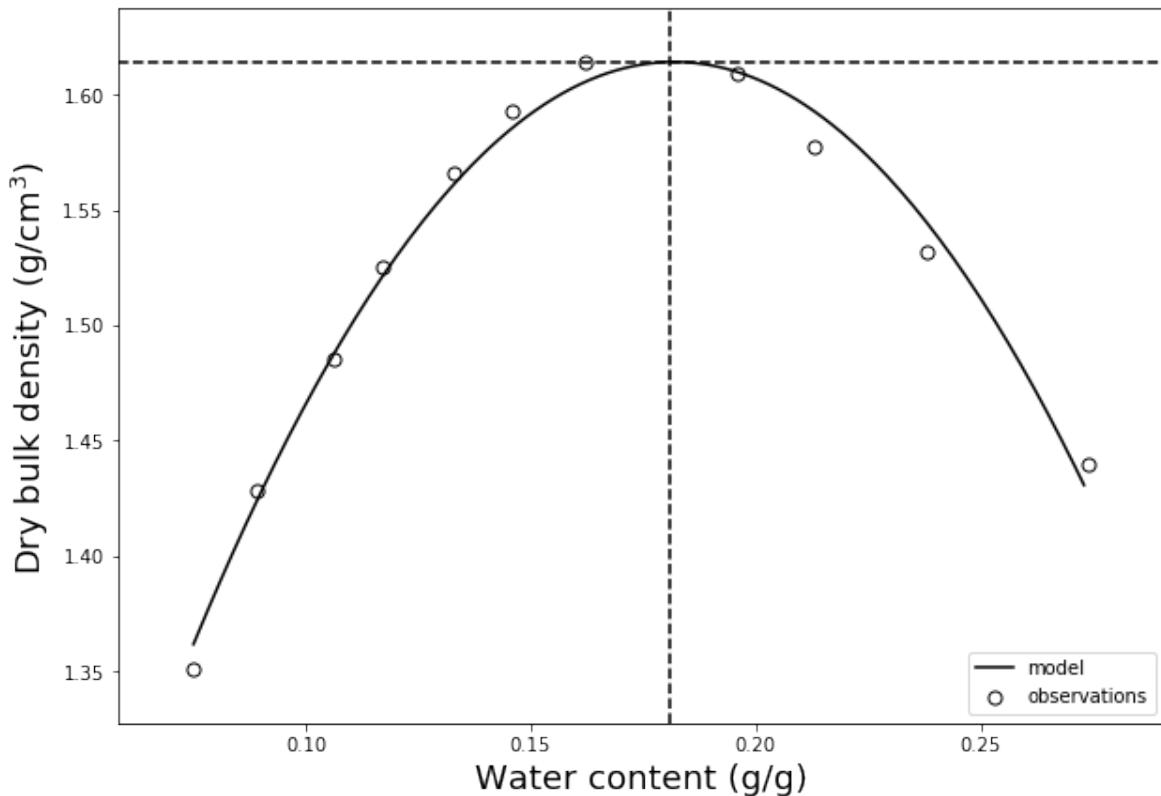
```

# Complete figure

water_min = np.min(data.gravimetric_water)
water_max = np.max(data.gravimetric_water)
water_pred = np.arange(water_min, water_max, 0.001)
dry_bulk_density_pred = polyfun(water_pred)

plt.figure(figsize=(10,7))
plt.scatter(data.gravimetric_water, data.dry_bulk_density, s=50, marker='o', facecolor='w',
            edgecolor='k', label='observed')
plt.plot(water_pred, dry_bulk_density_pred, '-k', label='model')
plt.axvline(optimal_water, color='k', linestyle='--')
plt.axhline(max_density, color='k', linestyle='--')
plt.xlabel('Water content (g/g)', size=18)
plt.ylabel('Dry bulk density (g/cm$^3$)', size=18)
plt.legend()
plt.show()

```



## 54.1 References

Davidson, J.M., Gray, F. and Pinson, D.I., 1967. Changes in Organic Matter and Bulk Density with Depth Under Two Cropping Systems 1. *Agronomy Journal*, 59(4), pp.375-378.

Kok, H., Taylor, R.K., Lamond, R.E., and Kessen, S. 1996. Soil Compaction: Problems and Solutions. Department of Agronomy. Publication no. AF-115 by the Kansas State University Cooperative Extension Service. Manhattan, KS. You can find the article at this link: <https://bookstore.ksre.ksu.edu/pubs/AF115.pdf>

Proctor, R., 1933. Fundamental principles of soil compaction. *Engineering news-record*, 111(13).

## 55 Counting seeds

In this exercise we will learn how to use image analysis to identify and count seeds from an image collected with a regular mobile device camera.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

from skimage.filters import threshold_otsu
from skimage.morphology import area_opening, disk, binary_closing
from skimage.measure import find_contours, label
from skimage.color import rgb2gray, label2rgb

# Read color image
image_rgb = mpimg.imread('../datasets/images/seeds.jpg')

# Convert image to grayscale
image_gray = rgb2gray(image_rgb)

# Visualize rgb and grayscale images
plt.figure(figsize=(10,6))

plt.subplot(1,2,1)
plt.imshow(image_rgb)
plt.axis('off')
plt.title('RGB')
plt.tight_layout()

plt.subplot(1,2,2)
plt.imshow(image_gray, cmap='gray')
plt.axis('off')
plt.title('Gray scale')
plt.tight_layout()
```

```
plt.show()
```



```
# Segment seeds using a global automated threshold
global_thresh = threshold_otsu(image_gray)
image_binary = image_gray > global_thresh

# Display classified seeds and grayscale threshold
plt.figure(figsize=(12,4))

plt.subplot(1,3,1)
plt.imshow(image_gray, cmap='gray')
plt.axis('off')
plt.title('Original grayscale')
plt.tight_layout()

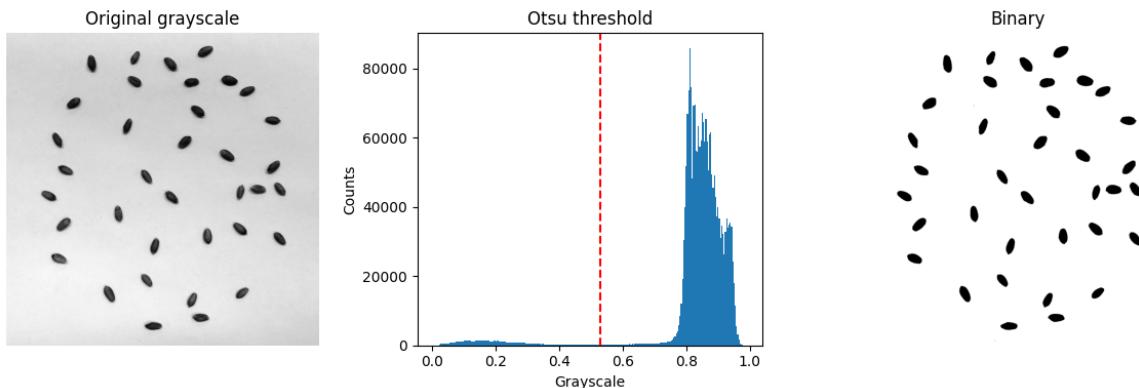
plt.subplot(1,3,2)
plt.hist(image_gray.ravel(), bins=256)
plt.axvline(global_thresh, color='r', linestyle='--')
plt.title('Otsu threshold')
plt.xlabel('Grayscale')
plt.ylabel('Counts')
```

```

plt.subplot(1,3,3)
plt.imshow(image_binary, cmap='gray')
plt.axis('off')
plt.title('Binary')
plt.tight_layout()

plt.show()

```



```

# Invert image
image_binary = ~image_binary

# Remove small areas (remove noise)
image_binary = area_opening(image_binary, area_threshold=1000, connectivity=2)

# Closing (performs a dilation followed by an erosion. Connect small bright patches)
image_binary = binary_closing(image_binary, disk(5))

# Let's inspect the structuring element
print(disk(5))

```

```

[[0 0 0 0 0 1 0 0 0 0 0]
 [0 0 1 1 1 1 1 1 1 0 0]
 [0 1 1 1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1 1 1 0]
 [1 1 1 1 1 1 1 1 1 1 1]
 [0 1 1 1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1 1 1 0]

```

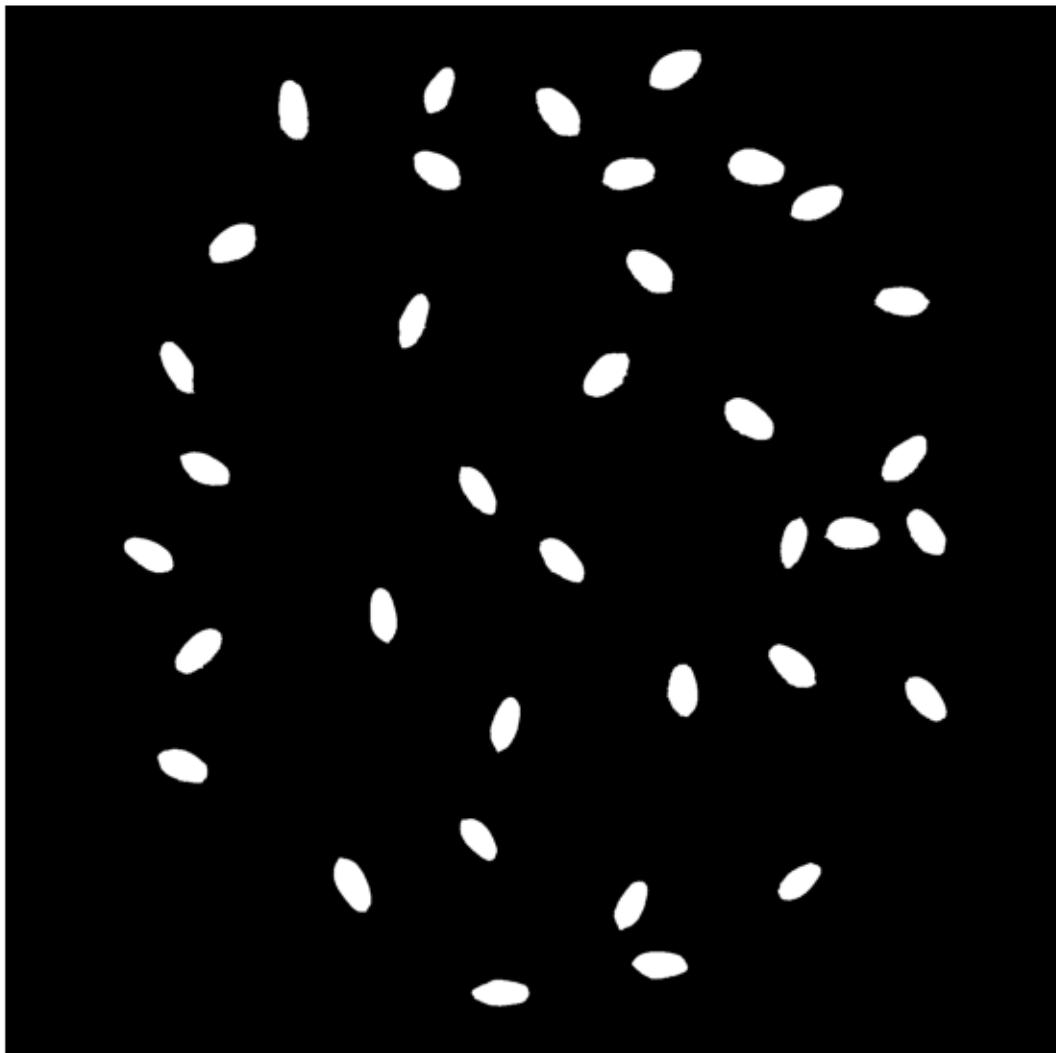
```
[0 1 1 1 1 1 1 1 1 1 0]
[0 0 1 1 1 1 1 1 1 0 0]
[0 0 0 0 0 1 0 0 0 0 0]]
```

```
# Display inverted and denoised binary image
plt.figure(figsize=(6,6))

plt.imshow(image_binary, cmap='gray')
plt.axis('off')
plt.title('Binary')
plt.tight_layout()

plt.show()
```

Binary



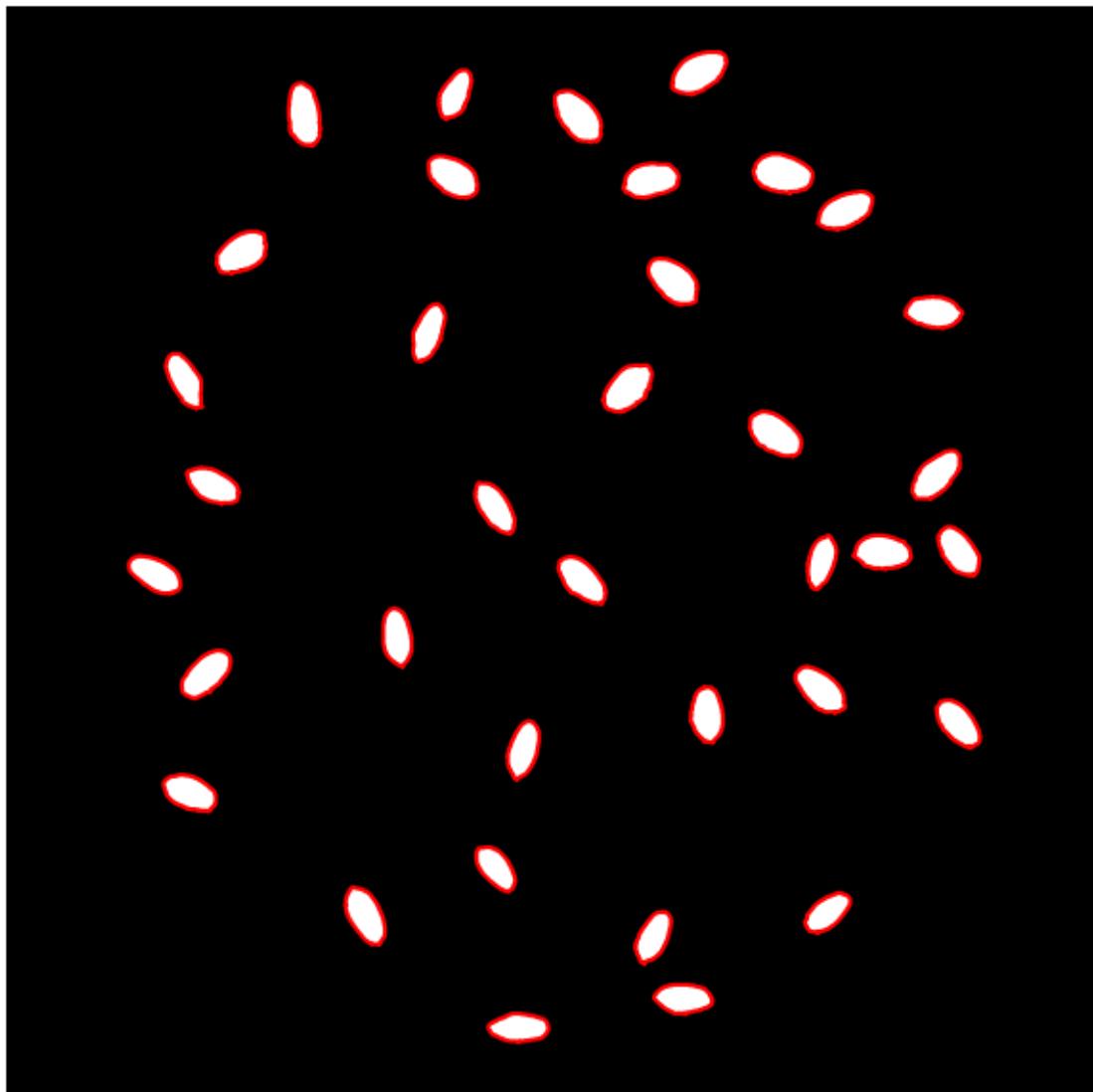
```
# Identify seed boundaries
contours = findContours(image_binary, 0)

# Print number of seeds in image
print('Image contains',len(contours),'seeds')
```

Image contains 36 seeds

```
# Plot seed contours
plt.figure(figsize=(6,6))
plt.imshow(image_binary, cmap='gray')
plt.axis('off')
plt.tight_layout()

for contour in contours:
    plt.plot(contour[:, 1], contour[:, 0], '-r', linewidth=1.5)
```



```
# Label image regions
label_image = label(image_binary)
image_label_overlay = label2rgb(label_image, image=image_rgb)

# Display image regions on top of original image
plt.figure(figsize=(6, 6))
plt.imshow(image_label_overlay)
plt.tight_layout()
plt.axis('off')
plt.show()
```

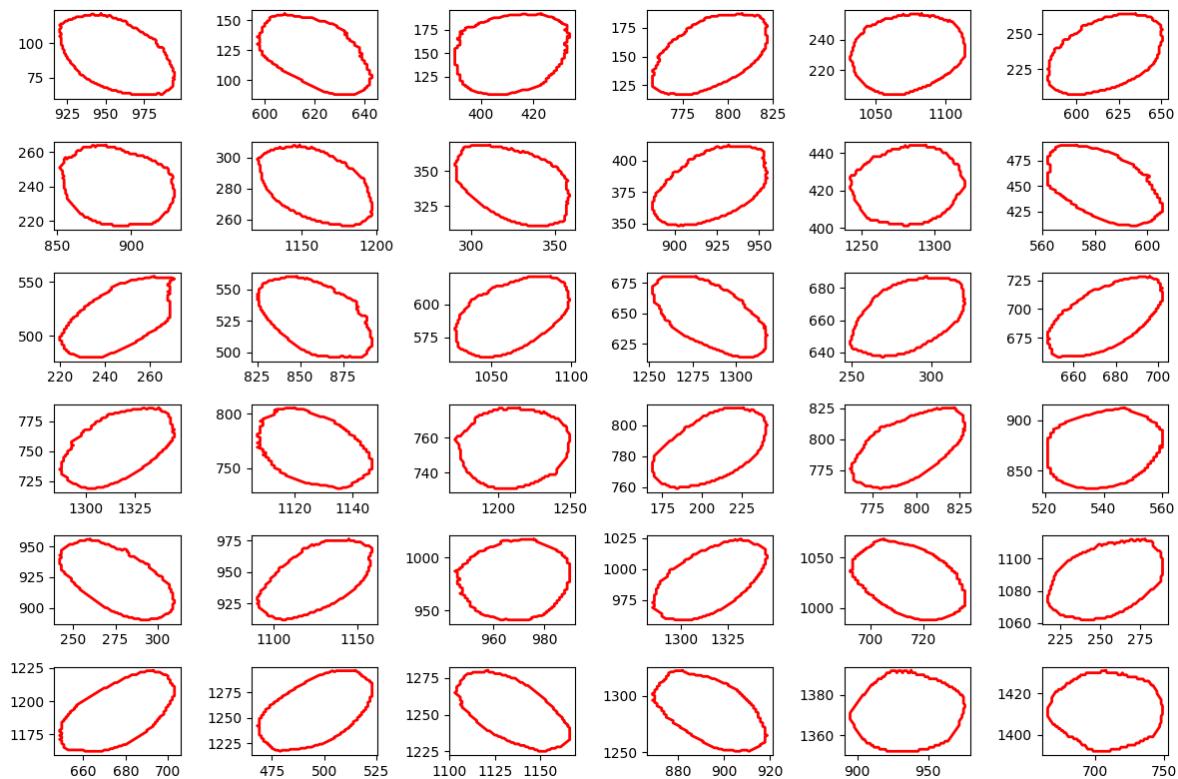


```

# Display contour for a single seed
plt.figure(figsize=(12, 8))

for seed in range(36):
    plt.subplot(6,6,seed+1)
    plt.plot(contours[seed] [:, 1], contours[seed] [:, 0], '-r', linewidth=2)
    plt.tight_layout()
plt.show()

```



# 56 Canopy cover

In this exercise we will learn how to quantify the percent of green canopy cover from downward-facing digital images collected using a point-and-shoot camera or mobile device.

The classification technique uses information in the Red, Green, and Blue (RGB) bands of the image. Each band consists of a 2D matrix of  $m$  rows by  $n$  columns.

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

## 56.1 Read and process a single image

```
# Read example image
rgb = mpimg.imread('../datasets/images/grassland.jpg')
```

```
# Display image
plt.imshow(rgb)
plt.axis('off')
plt.show()
```



```
# Inspect shape  
print(rgb.shape)  
print(rgb.dtype)
```

```
(512, 512, 3)  
float32
```

Images are often represented as unsigned integers of 8 bits. This means that each pixel in each band can only hold one of 256 integer values. Because the range is zero-index, the pixel values can range from 0 to 255. The color of a pixel is represented by triplet, for example the triplet (0,0,0) represents black, while (255,255,255) represents white. Similarly, the triplet (255,0,0) represents red and (255,220,75) represents a shade of yellow.

```
# Extract data in separate variable for easier manipulation.  
red = rgb[:, :, 0] #Extract matrix of red pixel values (m by n matrix)  
green = rgb[:, :, 1] #Extract matrix of green pixel values  
blue = rgb[:, :, 2] #Extract matrix of blue pixel values
```

```
# Compare shape with original image
print(red.shape)

(512, 512)

# Show information in single bands
plt.figure(figsize=(12,8))

plt.subplot(2,3,1)
plt.imshow(red, cmap="gray")
plt.axis('off')

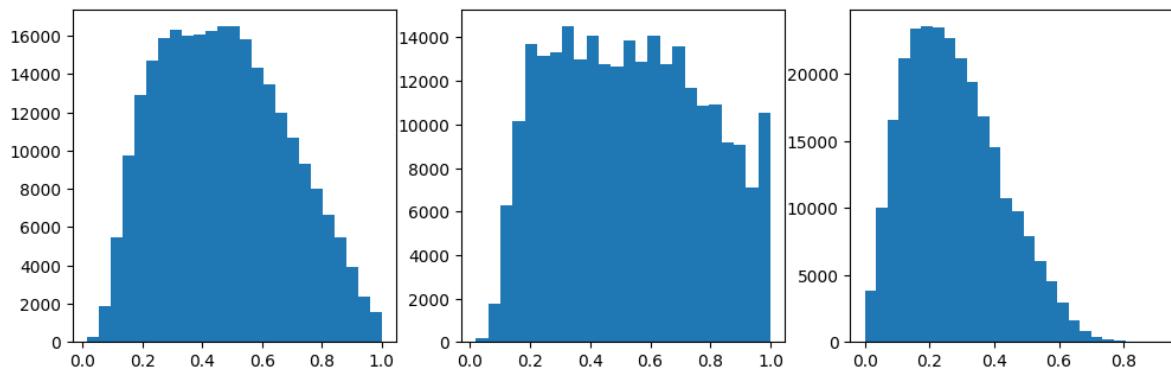
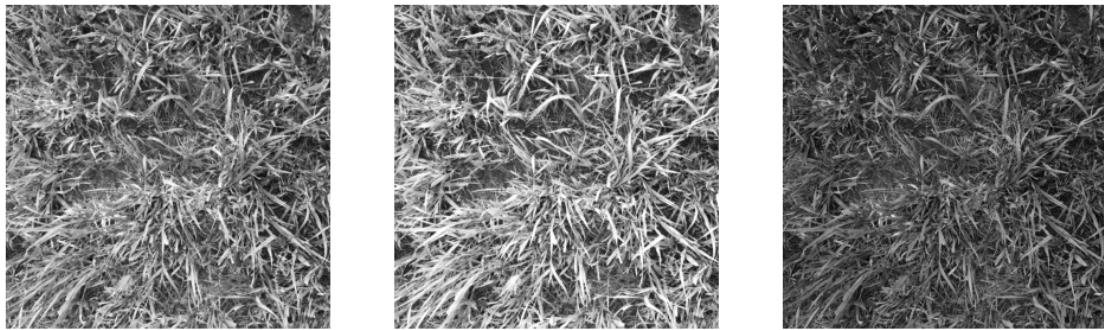
plt.subplot(2,3,2)
plt.imshow(green, cmap="gray")
plt.axis('off')

plt.subplot(2,3,3)
plt.imshow(blue, cmap="gray")
plt.axis('off')

# Add histograms using Doane's rule for histogram bins
plt.subplot(2,3,4)
plt.hist(red.flatten(), bins='doane')

plt.subplot(2,3,5)
plt.hist(green.flatten(), bins='doane')

plt.subplot(2,3,6)
plt.hist(blue.flatten(), bins='doane')
plt.show()
```



Find out more about all the different methods for generating hsitogram bins [here](#)

```
# Calculate red to green ratio for each pixel. The result is an m x n array.
red_green_ratio = red/green

# Calculate blue to green ratio for each pixel. The result is an m x n array.
blue_green_ratio = blue/green

# Excess green
ExG = 2*green - red - blue

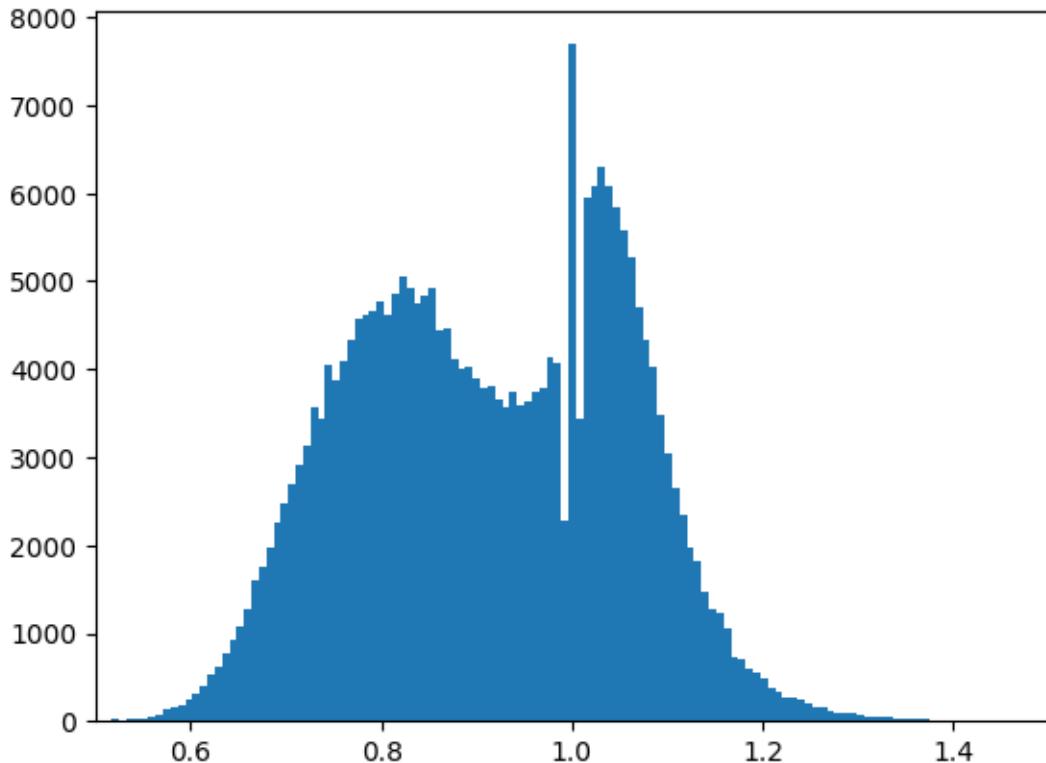
# Let's check the resulting data type of the previous computation
print(red_green_ratio.shape)
print(blue_green_ratio.dtype)

(512, 512)
float32
```

The size of the array remains unchanged, but Python automatically changes the data type

from `uint8` to `float64`. This is great because we need to make use of a continuous numerical scale to classify our green pixels. By generating the color ratios our scale also changes. Let's look at this using a histogram.

```
# Plot histogram
plt.figure()
plt.hist(red_green_ratio.flatten(), bins='scott')
plt.xlim(0.5,1.5)
plt.show()
```



```
# Classification of green pixels
bw = np.logical_and(red_green_ratio<0.95, blue_green_ratio<0.95, ExG>20)

print(bw.shape)
print(bw.dtype)
print(bw.size)
```

(512, 512)

```
bool  
262144
```

See that we started with an  $m \times n \times 3$  (original image) and we finished with and  $m \times n \times 2$  (binary or classified image)

```
# Compute percent green canopy cover  
canopy_cover = np.sum(bw) / np.size(bw) * 100  
print('Green canopy cover:', round(canopy_cover, 2), '%')
```

```
Green canopy cover: 56.64 %
```

```
plt.figure(figsize=(12,4))  
  
# Original image  
plt.subplot(1, 2, 1)  
plt.imshow(rgb)  
plt.title('Original')  
plt.axis('off')  
  
# Classified image  
plt.subplot(1, 2, 2)  
plt.imshow(bw, cmap='gray')  
plt.title('Classified')  
plt.axis('off')  
  
plt.show()
```



Classified pixels are displayed in white. The classification for this image is exceptional due to the high contrast between the plant and the background. There are also small regions where our approach misclassified green canopy cover as a consequence of bright spots on the leaves. For many applications this error is small and can be ignored, but this issue highlights the importance of taking high quality pictures in the field.

Tip

If possible, take your time to collect high-quality and consistent images. Effective image analysis starts with high quality images.

## 56.2 References

Patrignani, A. and Ochsner, T.E., 2015. Canopeo: A powerful new tool for measuring fractional green canopy cover. *Agronomy Journal*, 107(6), pp.2312-2320.

# 57 Cleaning yield monitor data

Yield monitor data is an excellent field diagnostic layer of information that provides insights about the high and low yield producing areas of the field. Yield maps can reveal spots that have low fertility values, high/low pH values, excessive water standing, soils with low water holding capacity that are vulnerable to dry periods, areas of the field affected by soil compaction due to traffic, or a combination of all the above.

The harvesting process is not simple and introduces spurious observations. This is affected by driving patterns, grain moisture, presence of weeds, field traffic condicitons, field shape, and field elevation changes are among the most common. Combine drivers have to maximize the time the combine is ingesting grain using the entire width of the platform, but during turns or at the beginning/end of the harvesting process this may not be achievable and thus it looks like the yield is lower in that part of the field. This is just an artifice and points out the need to clean yield monitor data before drawing any conclusions.

The yield monitor of modern combines collects data about combined speed, geographic position, time, grain flow, and grain moisture in real time at a temporal resolution of 1 second. This exorbitant temporal resolution generates rich datasets for spatial variability analysis.

## 57.1 Dataset description

In this exercise we will use a real yield monitor dataset from a soybean crop harvested in the state of Kansas in October of 2015. The farm and geogrpahic infromation were removed to preserve the farmers anonymity. So, the geographic coordinates were replaced by relative UTM coordinates. The X and Y coordinates are in meters relative to the center of the field (X=0, Y=0).

Units of variables in dataset:

- Flow in lbs/second
- Area in acres
- Distance in inches
- Duration in seconds
- Yield in lbs/acre
- Moisture in percent
- Width in inches

## 57.2 File formats

Yield monitor data is often saved using the Shapefile format (.shp). In this case I saved the file in .csv format to enable everyone to access the exercise using the pandas library.

For those interested in applying the same techniques to their own yield monitor data in .shp format, I recommend installing the geopandas library. The line below should get you started.

```
!pip install geopandas # Run in separate cell
df = gpd.read_file("../datasets/yield_monitor.shp")
df.head()
```

```
#import geopandas as gpd
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

df = pd.read_csv("../datasets/yield_monitor.csv")
print(df.shape)
df.head()
```

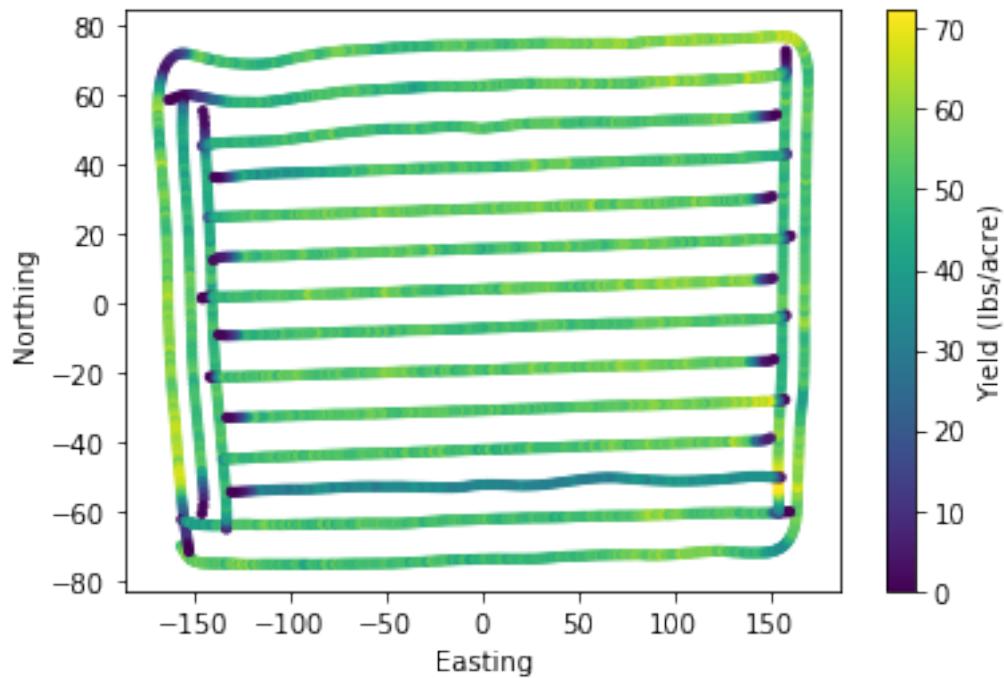
(3281, 12)

|   | Geometry | X           | Y          | Crop     | TimeStamp           | Yield | Flow   | Moisture | Durat |
|---|----------|-------------|------------|----------|---------------------|-------|--------|----------|-------|
| 0 | Point    | -156.576874 | -69.681188 | Soybeans | 2015-10-10 15:07:35 | 50.51 | 5.6990 | 13.0     | 1.0   |
| 1 | Point    | -156.100303 | -70.083748 | Soybeans | 2015-10-10 15:07:36 | 51.41 | 5.8003 | 13.0     | 1.0   |
| 2 | Point    | -155.632047 | -70.508652 | Soybeans | 2015-10-10 15:07:37 | 50.66 | 5.7151 | 13.0     | 1.0   |
| 3 | Point    | -155.198382 | -70.945248 | Soybeans | 2015-10-10 15:07:38 | 54.20 | 5.8286 | 13.0     | 1.0   |
| 4 | Point    | -154.808573 | -71.360394 | Soybeans | 2015-10-10 15:07:39 | 55.86 | 5.5141 | 13.0     | 1.0   |

```
# Convert dates to Pandas datetime format
df["TimeStamp"] = pd.to_datetime(df["TimeStamp"], format="%Y/%m/%d %H:%M:%S")

# Compute speed in miles per hour (mph)
df["Speed"] = df["Distance"] / df["Duration"] # in inches/second
df["Speed"] = df["Speed"]/63360*3600 # convert to miles/hour
```

```
# Examine data
plt.scatter(df["X"], df["Y"], s=10, c=df["Yield"])
plt.xlabel('Easting')
plt.ylabel('Northing')
plt.colorbar(label="Yield (lbs/acre)")
plt.show()
```



An extremely useful habit is to plot the data. Histograms are great at describing the central tendency and dispersion of a given variable in a single figure. Based on histograms we can select and fine-tune the variables and thresholds that we will use to denoise our yield monitor data.

```
plt.figure(figsize=(10,8))
plt.subplot(2,2,1)
plt.hist(df["Yield"], bins=13)
plt.xlabel("Yield (lbs/acre)", size=16)

plt.subplot(2,2,2)
plt.hist(df["Flow"])
plt.xlabel("Flow (lbs/second)", size=16)
```

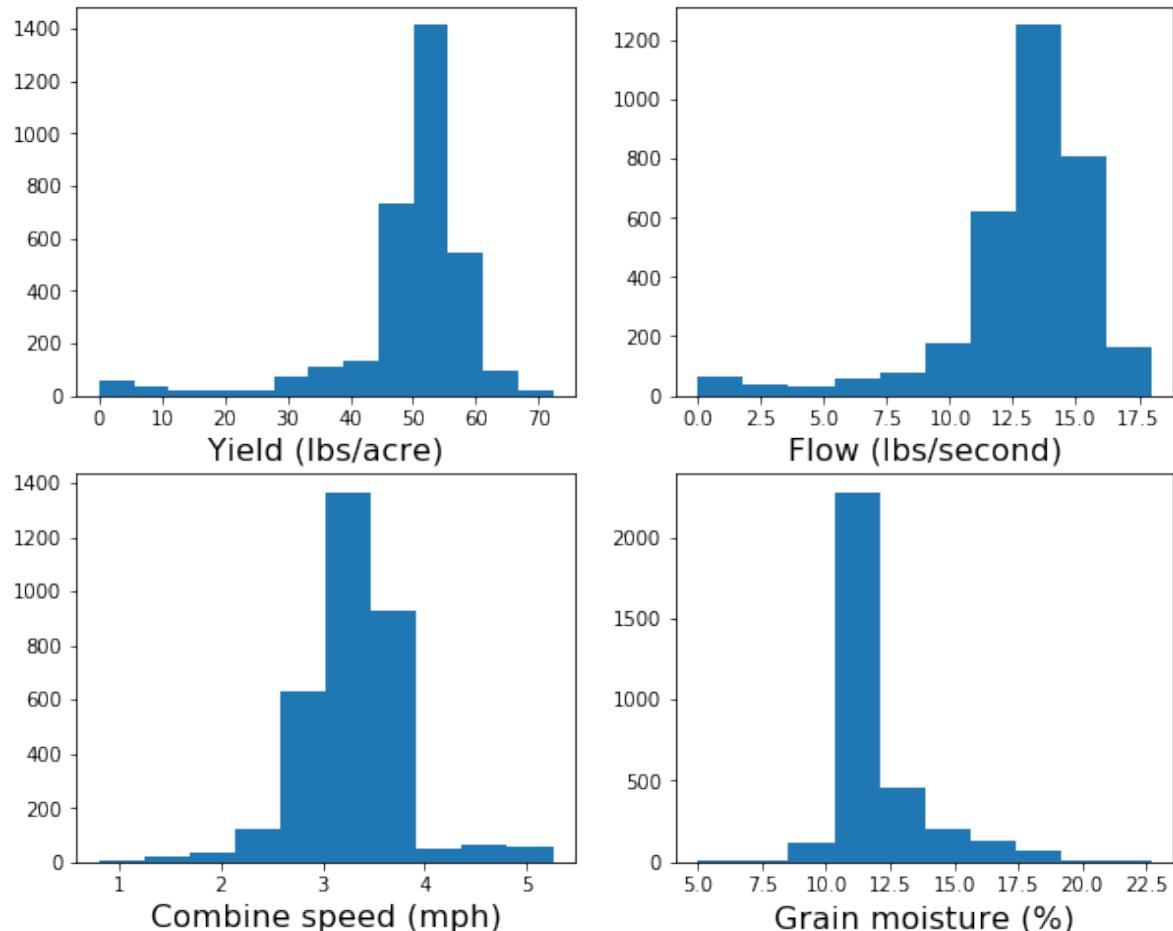
```

plt.subplot(2,2,3)
plt.hist(df["Speed"])
plt.xlabel("Combine speed (mph)", size=16)

plt.subplot(2,2,4)
plt.hist(df["Moisture"])
plt.xlabel("Grain moisture (%)", size=16)

plt.show()

```



```

# Create copy of original DataFrame
df_clean = df

```

### 57.3 Filtering rules with clear physical meaning

```
idx_too_slow = df["Speed"] < 2.5
idx_too_fast = df["Speed"] > 4
idx_too_wet = df["Moisture"] > 20
idx_too_dry = df["Moisture"] < 10
idx_low_flow = df["Flow"] <= 10
idx_high_flow = df["Flow"] >= 18
idx = idx_too_slow | idx_too_fast | idx_too_dry | idx_too_wet | idx_low_flow | idx_high_flow
df_clean = df[~idx]
```

Up to here, the method will probably clean most maps from yield monitors. If you want to stop here I also suggest adding the following two boolean conditions to filter out outliers. If you decide to implement a more sophisticated approach, then you can probably omit the next two lines since we will be implementing a moving filter that will capture outliers later on.

```
idx_high_yield = df["Yield"] >= df["Yield"].quantile(0.95)
idx_low_yield = df["Yield"] <= df["Yield"].quantile(0.05)
```

```
print(df.shape)
print(df_clean.shape)
removed_points = df.shape[0] - df_clean.shape[0]
print("Removed", str(removed_points), 'points')
```

```
(3281, 13)
(2760, 13)
Removed 521 points
```

Let's generate a plot showing the original and resulting dataset after our first layer of cleaning. We will circle the points that we removed to check our work.

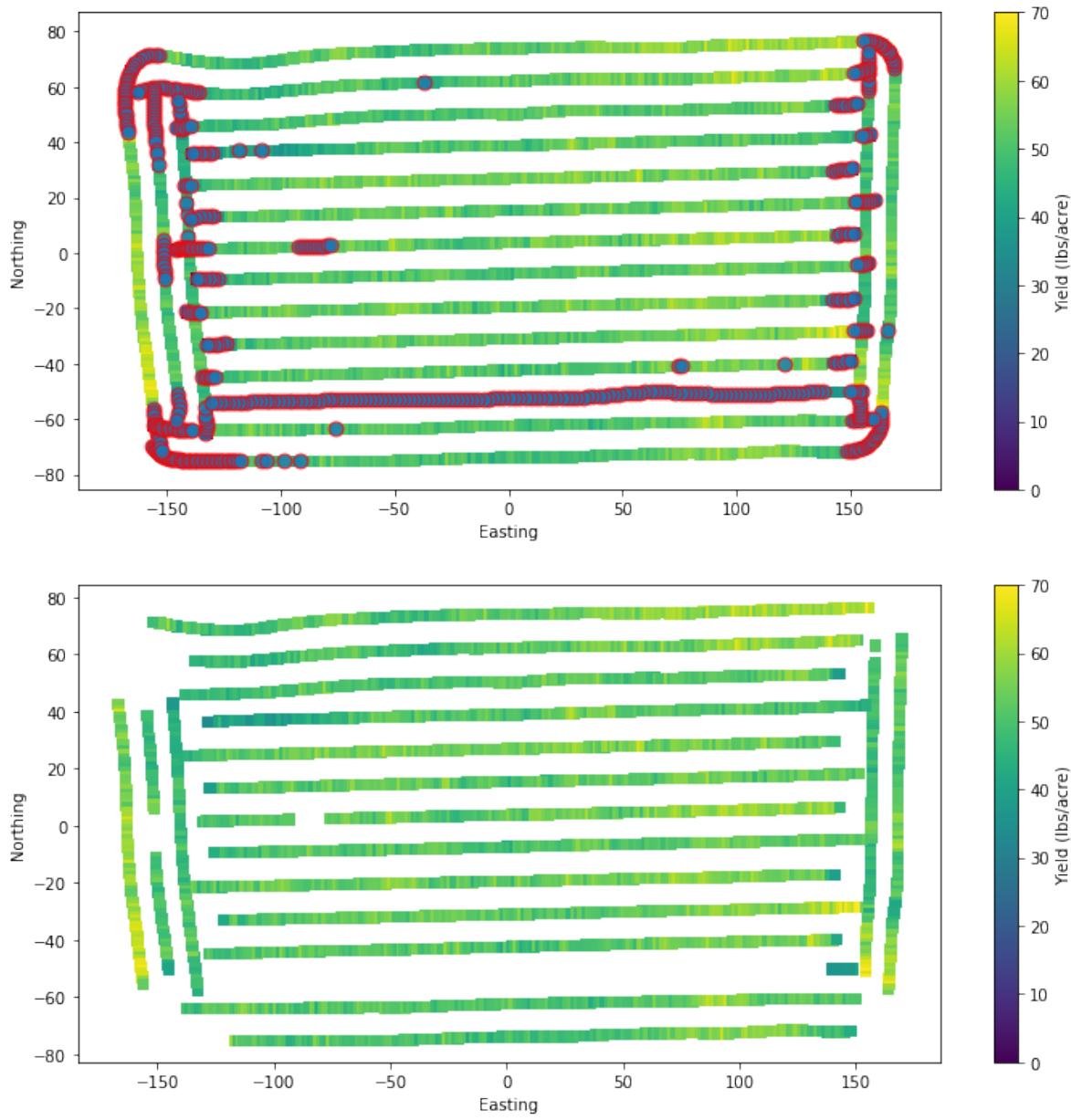
```
plt.figure(figsize=(12,12))

plt.subplot(2,1,1)
plt.scatter(df["X"], df["Y"], s=40, marker='s', c=df["Yield"])
plt.colorbar(label="Yield (lbs/acre)")
plt.ylim(0,70)
plt.scatter(df.loc[idx,"X"], df.loc[idx,"Y"],
            s=80,
            marker='o',
```

```
    facecolor=None,
    edgecolor='r',
    alpha=0.8)
plt.xlabel('Easting')
plt.ylabel('Northing')

plt.subplot(2,1,2)
plt.scatter(df_clean["X"], df_clean["Y"], s=40, marker='s', c=df_clean["Yield"])
plt.colorbar(label="Yield (lbs/acre)")
plt.clim(0,70)
plt.xlabel('Easting')
plt.ylabel('Northing')

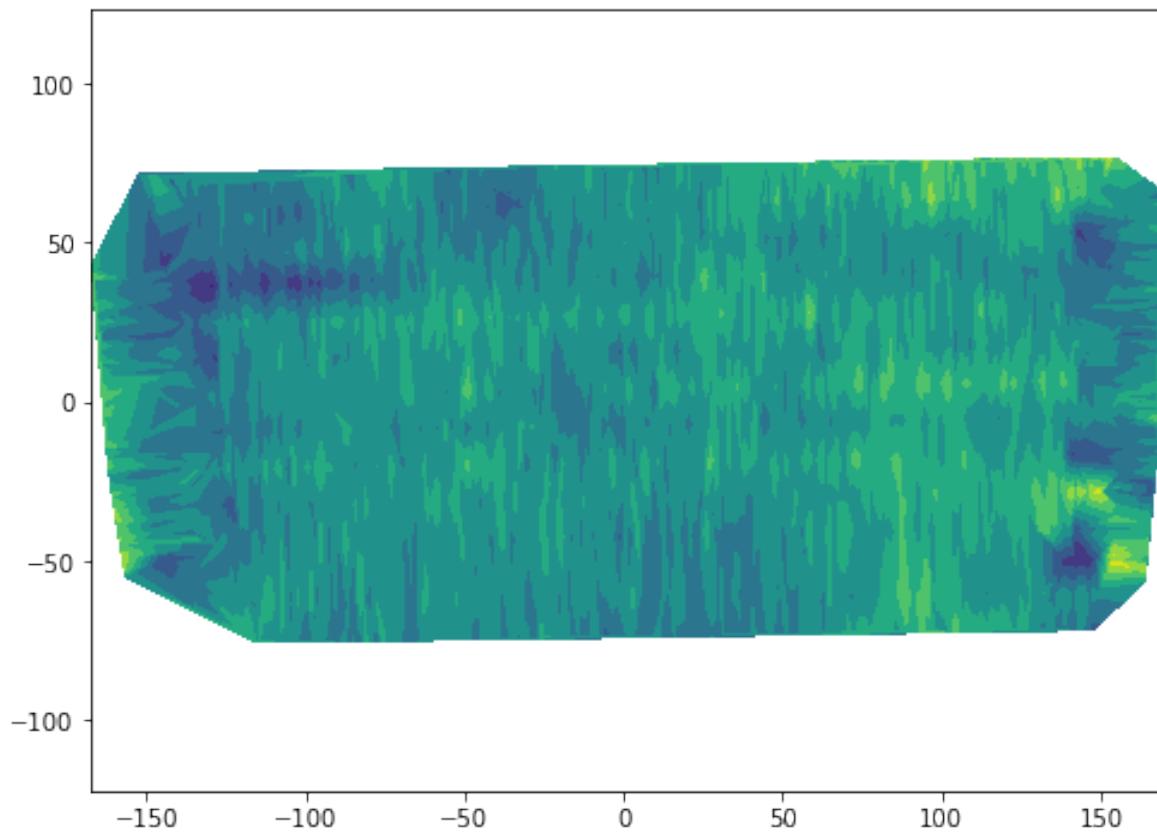
plt.show()
```



```
# Remove NaNs
df_clean = df_clean.dropna()

plt.figure(figsize=(8,6))
plt.tricontourf(df_clean["X"], df_clean["Y"], df_clean["Yield"], levels=7)
```

```
plt.axis('equal')
plt.show()
```



```
# Reset index
df_clean.reset_index(drop=True, inplace=True)
df_clean.head()
```

|   | Geometry | X           | Y          | Crop     | TimeStamp           | Yield | Flow    | Moisture | Dura |
|---|----------|-------------|------------|----------|---------------------|-------|---------|----------|------|
| 0 | Point    | -116.700755 | -75.081549 | Soybeans | 2015-10-10 15:08:21 | 59.83 | 12.1293 | 10.2     | 1.0  |
| 1 | Point    | -115.518013 | -75.072397 | Soybeans | 2015-10-10 15:08:22 | 49.25 | 10.1585 | 10.4     | 1.0  |
| 2 | Point    | -114.387442 | -75.064138 | Soybeans | 2015-10-10 15:08:23 | 56.08 | 11.0729 | 10.9     | 1.0  |
| 3 | Point    | -113.230785 | -75.055433 | Soybeans | 2015-10-10 15:08:24 | 51.03 | 10.2547 | 11.2     | 1.0  |
| 4 | Point    | -112.073939 | -75.057824 | Soybeans | 2015-10-10 15:08:25 | 51.44 | 10.4287 | 11.5     | 1.0  |

## 57.4 Moving filters

A moving filter is usually a sliding window that performs a smoothing operation. This usually works great with regular grids, but in the case of yield monitor data we deal with irregular grids, which requires that we handle the window in a slightly different way.

We will iterate over each point collected by the combine and at each iteration step we will find the observations within a given distance from the combine and we will either:

- Assign the current point the median value of all its neighboring points within a specific distance radius.
- Use the yield value of all the selected neighboring points to determine whether the value of the current point is within the 5 and 95th percentile of the local yields.

Certainly there are many other options. I just came up with these two simple approaches based on previous studies and my own experience.

In any case, we need to be able to compute the distance from the current point in the iteration to all the other points. This is how we will determine the neighbors.

### 57.4.1 Function to compute Euclidean distance

```
def edist(xpoint, ypoint, xvec, yvec):
    """Compute and sort Euclidean distance from a point to all other points."""
    distance = np.sqrt((xpoint - xvec)**2 + (ypoint - yvec)**2)
    idx = np.argsort(distance)

    return {"distance":distance, "idx":idx}
```

### 57.4.2 Moving median filter

```
df_medfilter = df_clean

for i in range(df_medfilter.shape[0]):

    # Compute Euclidean distance from each point to the rest of the points
    D = edist(df_medfilter["X"][i], df_medfilter["Y"][i], df_medfilter["X"], df_medfilter["Y"])

    # Find index of neighbors
```

```

idx = D["distance"] <= 5 # meters

# Replace current value with median of neighbors
df_medfilter.loc[i,"Yield"] = df_medfilter.loc[idx,"Yield"].median()

# This can take a while, so let's print something to know that the interpreter is done.
print('Done!')

```

Done!

```

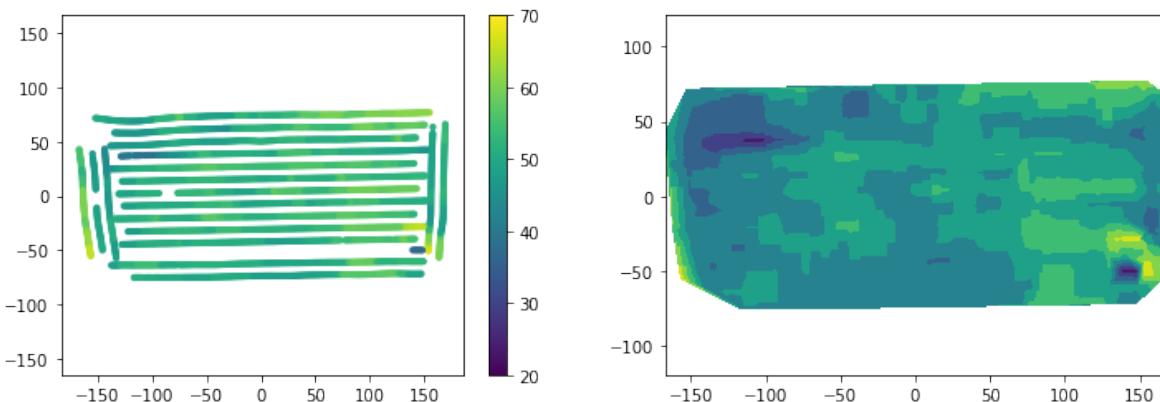
plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
plt.scatter(df_medfilter["X"], df_medfilter["Y"], s=10, c=df_medfilter["Yield"])
plt.colorbar()
plt.clim(20, 70)
plt.axis('equal')

plt.subplot(1,2,2)
plt.tricontourf(df_medfilter["X"], df_medfilter["Y"], df_medfilter["Yield"])
plt.axis('equal')

plt.show()

```



### 57.4.3 Moving filter to detect outliers

```
df_outfilter = df_clean.copy()

for i in range(df_outfilter.shape[0]):

    # Compute Euclidean distance from each point to the rest of the points
    D = edist(df_outfilter["X"][i], df_outfilter["Y"][i], df_outfilter["X"], df_outfilter["Y"])

    idx = D["distance"] <= 5 # meters
    current_point_yield = df_outfilter.loc[i,"Yield"]

    # Find lower and upper threshold based on percentiles
    Q5 = df_outfilter.loc[idx,"Yield"].quantile(0.05)
    Q95 = df_outfilter.loc[idx,"Yield"].quantile(0.95)

    # If current point is lower or greater than plausible neighbor values, then set to NaN
    if (current_point_yield < Q5) | (current_point_yield > Q95):
        df_outfilter.loc[i,"Yield"] = np.nan

print("Done!")
```

Done!

Check if your method detected any outliers, which should have been assigned `NaN` to the yield variable.

```
df_outfilter["Yield"].isna().sum()
```

392

There are some `NaN` values. The interpolation method `tricontourf` does not handle `NaN`, so we need to remove them from the DataFrame first. We will use the Pandas `dropna()` to do this easily.

```
df_outfilter = df_outfilter.dropna()
df_outfilter["Yield"].isna().sum()
```

0

Perfect, our DataFrame no longer contains NaN values. Before we proceed, let's also check the difference in the number of points between the two methods. Just to know whether our filtering was a bit excessive. If it was, then realexed some of the initial paramters or the quantiles threshlds.

```
print(df_medfilter.shape)
print(df_outfilter.shape)

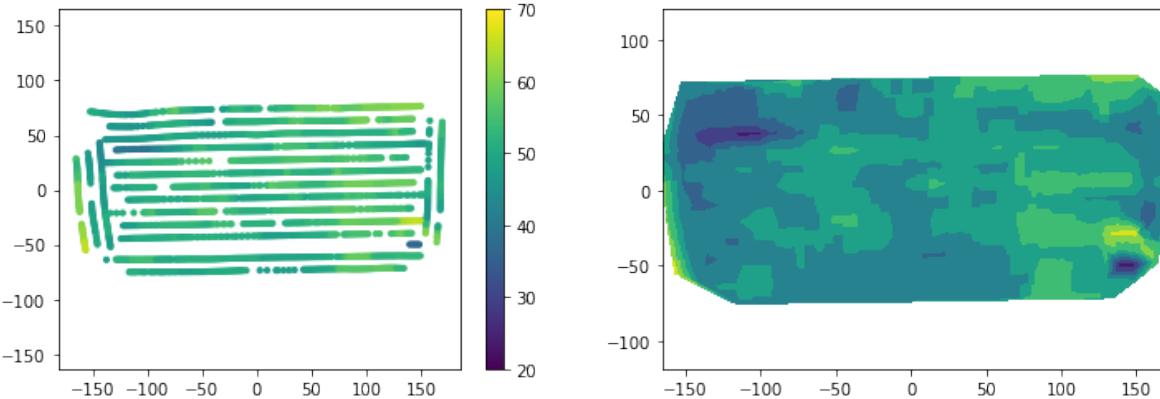
(2760, 13)
(2368, 13)

plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
plt.scatter(df_outfilter["X"], df_outfilter["Y"], s=10, c=df_outfilter["Yield"])
plt.colorbar()
plt.clim(20, 70)
plt.axis('equal')

plt.subplot(1,2,2)
plt.tricontourf(df_outfilter["X"], df_outfilter["Y"], df_outfilter["Yield"])
plt.axis('equal')

plt.show()
```



## **57.5 Observations**

The approaches tested in exercise yielded somewhat similar results. The first layer of outliers removal based on the plausible value of variables with clear physical meaning such as combine speed, grain flow rate, and grain moisture content are an effective way of removing clear outliers.

The first layer does not result in a clear interpolation, at least using the `tricontourf` function. I'm sure that some of the filters with scipy and image processing toolboxes can solve this issue without further processing.

A median filter is a powerful filter widely used in image analysis and was effective to remove outliers and dramatically improved the map in terms of smoothness and visual patterns. This method does not preserve the original values.

The moving window to remove outliers based on quantile thresholds performed similar to the median filter and represents an alternative method. This method preserves the original values and removes values that are considered outliers.

The method of choice depends on the user, the complexity of the data, and the performance of the methods compared to known field patterns and observations during harvest.

## **57.6 References**

Khosla, R. and Flynn, B., 2008. Understanding and cleaning yield monitor data. Soil Science Step-by-Step Field Analysis, (soilsciencesstep), pp.113-130.

Kleinjan, J., Chang, J., Wilson, J., Humburg, D., Carlson, G., Clay, D. and Long, D., 2002. Cleaning yield data. SDSU Publication.