Session 1. Basics of working with `Python`

2026-02-12

Highlights:

This is my mini-reflection. Paragraphs must be indented.
It can contain multiple paragraphs.

"Be willing to be a beginner every single morning."

—– Meister Eckhart

Session outline

- Why `Python`?
- Installing the software: Python and RStudio (with reticulate)
- Packages (libraries)
- Installing packages: PyPI, Conda, and other sources
- Getting help
- Creating a project
- Directory structure
- Creating new files: types of files
- Literate programming
- Data objects/classes and basic operations

Introduction

NOTE: This is an Quarto Markdown document. This type of document is a plain text file that can recognize chunks of code. When you execute code within the document, the results are displayed beneath. Quarto Markdown files are *computational notebooks* which implement a coding philosophy called *literate programming*. Literate computing emphasizes the use of natural language to communicate with humans and chunks of code to communicate with the computer. By making the main audience other humans, this style of coding flips around the usual way in which code is written (computer is main audience, humans come second). This helps to make learning how to code more intuitive and accessible.

Let us first explain the anatomy of an Quarto Markdown document.

At the top of the document is a *header* (called a YAML header) contained between two sets of three dashes `---`. The header includes metainformation about the document, including possibly the title of the document, the name(s) of the author(s), and how to process the document (e.g., what type of output to produce). After the header comes the *body* of the document, that is, the main contents of the document (text and code).

The header is quite an important compontent of the document, but we will for the moment take is as given and concentrate on the body of the document. This file, for example, introduces some basic instructions to work with `Python` and Quarto Markdown.

Chunks of code are key to writing computational notebooks. Whenever you see a chunk of code in an Quarto Markdown document as follows, you can run it (by clicking the green 'play' icon on the top right corner of the code window) to see the results. Try it below!

```python
print("Hello, Workshop")
```

```
Hello, Workshop
```

You can see that the function `print()` displays the argument on the screen.

Notice the way the document speaks to you in natural language, and to the computer in `Python`. The computer, instead of being the focus of the document, is an assistant to illustrate concepts (like, what is a chunk of code?).

Each of the documents in this workshop (the *Sessions*) is like a set of lecture notes. You can *knit* the document to produce a PDF file to study. The documents, on the other hand, are potentially much more than simple lecture notes, and they can in fact become the foundation for your own experiments and annotations. As you read and study, you can 'customize' the notes based on your developing understanding of the subject matter and/or to complement the document with other examples and information. To make the readings uniquely yours, you can type directly into the document (the original template is still available from the package, and if you need a fresh start, you can always create a new file with it).

In addition, you can use the following style to create a *textbox*:

NOTICE:
This is an annotation. Here I can write my thoughts as I study, or I can add useful links or other information to help me learn.
To create a new paragraph, I need to type two blanks after the last one.

A textbox allows you to highlight important information. Try creating your own textbox next.

By now you will already have noticed a few conventions for writing in R Markdown:

- A chunk of Python code begins with three backticks ("") and the letter 'python' between curly brackets; a chunk of code concludes with three backticks.
- Hashtags ("#") indicate headers: one hashtag is a primary header, two hashtags is a secondary header, three a tertiary header, etc.
- Asterisks are used for changing the font to *Italics*, **Bold**, and ***Italics+Bold***.
- Superscrit can be added by using `^`, such as in superscript^2
- Subscript can be added by using `~`, such as in superscript$_2$
- Dashes are used to create unnumbered lists.
- Hyperlinks can be introduced by using square brackets followed by the url in brackets.
- Greater-than (">") is used for block-quotes.

There are some other typing conventions that you can find in this useful cheatsheet.

RStudio Window

If you are reading this, you must already have learned how to create a new project and documents using the templates in the workshop package. We can now proceed to discuss some basic concepts regarding data types and operations.

Preliminaries

We will load all *libraries* we plan to use in this session. A library (or package) is a collection of modules containing pre-written code, functions, and classes that developers can use to perform common tasks without writing code from scratch. These are shareable code units that extend base `Python`'s capabilities. When you install `Python`, it includes built-in standard libraries for mathematical operations, date/time handling, system interactions, and more.

However, sometimes we need to perform analyses for which no built-in libraries exist. In these cases, we must install libraries from external sources hosting package repositories or "indexes." The two most common are `PyPI`(Python Package Index) and `Conda` repositories, but packages can also come from `GitHub` repositories and other locations.

`Conda`'s advantage is its dependency tracking-it manages how libraries depend on one another, reducing compatibility issues.

For `PyPI` packages, use `pip install package_name` in the terminal. For `Conda`, use `conda install package_name`.

Two libraries will be extensively used in this workshop: `matplotlib` and `pandas`. `Matplotlib` is `Python`'s most popular data visualization library, enabling the creation of 2D plots. `Pandas` provides robust data structures and functions for working with tabular data, primarily Series and DataFrames.

If you haven't already, install these packages with `pip install pandas matplotlib` or `conda install pandas matplotlib`. Once installed, you must load them into memory using `import package_name`:

```
import pandas as pd
```

Language Semantics

Python uses indentation (spaces or tabs) to structure code, unlike languages like `R` and `Java`, which use curly braces (`{}`).

Text following the `#` symbol is treated as a comment and ignored by the Python interpreter:

```
data # Any comment
# Another comment
```

As in other languages, `Python` functions are reusable code blocks that perform specific actions when called. We call them by name followed by parentheses (`()`). Parameters can be included within the parentheses. For example, a function `f` with parameters `a`, `b`, and `c` is called as:

```
f(a, b, c)
```

In `Python`, everything is an *object.* Every data type is a `Python` object with a specific type, internal data, and associated functions. Functions within objects are called *methods*, providing access to the object's content. For example, if we have an object data and want to access information via its method:

```
data.method()
```

Objects also have *attributes*: values representing properties or characteristics. Like methods, attributes are accessed using dot notation:

```
data.attribute
```

`Python` uses `=` to assign values to variables. Variables are symbolic names that reference objects or values stored in memory:

```
var = 10
var
```

```
10
```

In the chunk above, we assigned the value 10 to the variable `var.` This assignment allows us to retrieve the value later.

Data types in `Python`

After executing the previous chunk, you'll notice a variable named var appears in your `Environment` (upper-right pane tab). This variable holds a single value (10) and belongs to a group of data types known as scalars, which store single values. Standard built-in scalar types are:

Table 1: Standard scalars in Python.

Type	Description
int	Integer (whole numbers)
float	Floating-point numbers, to represent real numbers with a decimal point.
bool	A boolean value. True or False.
bytes	Pure Bytes ASCII
None	Null value
str	String type, to represent text data.

You can check a variable's data type using the `type()` function:

```python
type(var)
```

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

Next, we'll explore these scalar data types in more detail.

Scalars

Numeric (`int` and `float`)

The main numeric types in Python are `int` and `float`:

```python
1
```

```
1
```

```python
type(1)
```

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

```python
3.141593
```

```
3.141593
```

```python
type(3.141593)
```

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

Boolean

Boolean values are `True` and `False`:

```python
False
```

```
False
```

```
type(False)
```

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

You can perform Boolean comparisons using `and`, `or`, and comparison operators (`==` for equal, `!=` for not equal, `>` for greater than, `>=` for greater than or equal, `<` for less than, `<=` for less than or equal):

```
4 > 3
```

```
True
```

```
2 == 3
```

```
False
```

```
5 == 2
```

```
False
```

```
4 >= 3
```

```
True
```

Strings

Strings represent text and are created using quotation marks (either " or '):

```
"This is a string"
```

```
'This is a string'
```

Numbers within quotation marks are stored as strings:

```
type("5")
```

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

Multi-line strings use triple quotes:

```
"""
This
is
an
example
of
a
string
with
multiple
lines
"""
```

```
'\nThis \nis \nan \nexample\nof \na \nstring\nwith\nmultiple \nlines\n'
```

The `\n` character indicates line breaks in text.

None

`None` represents null value in `Python`:

```
None
```

Built-in data structure

Now, consider storing multiple values in a variable. For this, you need data structures beyond scalars. We'll explore `Python`'s three built-in data structures: `tuples`, `lists`, and `dictionaries.`

Tuple

A tuple is a finite, ordered collection of items. Create a tuple by separating items with commas:

```
a_tuple = 100, 150, 45
a_tuple
```

```
(100, 150, 45)
```

You can create nested tuples:

```
nestedtuple = (18, 19, 20), (5, 4)
nestedtuple
```

```
((18, 19, 20), (5, 4))
```

You can also create tuples using the `tuple()` function:

```
tuple([2, 5, 10])
```

```
(2, 5, 10)
```

Tuples can store different scalar types:

```
tuple([2, 5, 10, "b"])
```

```
(2, 5, 10, 'b')
```

You can create tuples from any iterable sequence:

```
tuple("example")
```

```
('e', 'x', 'a', 'm', 'p', 'l', 'e')
```

Tuple elements have fixed positions and are immutable after creation. This structure is useful for storing related data of different types efficiently, such as coordinates (x, y) or user information (name, age, ID).

List

Lists are similar to vectors in other languages but can store different data types within the same list.
Create a list using square brackets:

```
b_list = [45, 50, 55, 60]
b_list
```

```
[45, 50, 55, 60]
```

Create a list using square brackets:

```
a_list = list(a_tuple)
a_list
```

```
[100, 150, 45]
```

Unlike tuples, lists are mutable. You can append items:

```
a_list.append(200)
a_list
```

```
[100, 150, 45, 200]
```

Insert items at specific positions:

```
b_list.insert(0, 40) # Add 40 at position 0 (first position)
b_list
```

```
[40, 45, 50, 55, 60]
```

Remove items by index using `pop()`:

```
b_list.pop(0) # Removing the value 40 (index 0) from the list
```

```
40
```

```
b_list
```

```
[45, 50, 55, 60]
```

Remove specific items by value using `remove()`:

```
b_list.remove(50)
b_list
```

```
[45, 55, 60]
```

Dictionaries

A dictionary (`dict`) is a collection of key-value pairs with flexible size, where keys and values are Python objects. Create a dictionary using curly braces with key-value pairs separated by colons:

```
fruits_dict = {'orange': 10, "banana":[5,10,20]}
fruits_dict
```

```
{'orange': 10, 'banana': [5, 10, 20]}
```

Add new key-value pairs:

```
fruits_dict["lemon"] = [40, 60, 80]
fruits_dict
```

```
{'orange': 10, 'banana': [5, 10, 20], 'lemon': [40, 60, 80]}
```

Access values by key:

```
fruits_dict["banana"]
```

```
[5, 10, 20]
```

Remove a key-value pair using pop():

```
fruits_dict.pop("banana")
```

```
[5, 10, 20]
```

```
fruits_dict
```

```
{'orange': 10, 'lemon': [40, 60, 80]}
```

List all keys in a dict:

```
fruits_dict.keys()
```

```
dict_keys(['orange', 'lemon'])
```

List all values in a dict:

```
fruits_dict.values()
```

```
dict_values([10, [40, 60, 80]])
```

Indexing

Indexing in Python accesses individual items in sequences (strings, lists, tuples) by their position. Python uses zero-based indexing: the first element is at index 0, the second at 1, and so on. Use square brackets (`[]`) after the sequence name.

Two indexing approaches:

- Positive indexing: Counts from the beginning (starting at 0).
- Negative indexing: Counts from the end (starting at -1 for the last item).

Retrieve the third character of a string:

```
"This is a string"[2]
```

```
'i'
```

Indexing also works for tuples:

```
a_list
```

```
[100, 150, 45, 200]
```

```
a_list[0] # the indexes starts on 0!
```

```
100
```

```
a_list[1]
```

```
150
```

```
a_list[2]
```

```
45
```

And for lists:

```
b_list[-1]
```

```
60
```

For dictionaries with multiple values per key, access specific values using indexing within the values:

```
fruits_dict['lemon'][0] # for the values value
```

```
40
```

```
fruits_dict['lemon'][0:2]
```

```
[40, 60]
```

Series and DataFrames

Other useful data structures in Python are Series and DataFrames, included in the `pandas` package. A Series is an array-like object containing a sequence of values and an associated array of labels called an index:

```
materials = pd.Series(["pens", "pencils", "paper", "notebook"])
materials
```

```
0        pens
1     pencils
2       paper
3    notebook
dtype: object
```

We create a Series by calling `pd.Series()` with the elements inside parentheses. The materials Series contains school supplies with indexes on the left. Since we didn't specify index values, pandas created integer indexes starting from 0.

Access Series values by index:

```
materials[0] # For one value
```

```
'pens'
```

```python
materials[0:2] # For more than one value, on a sequence
```

```
0       pens
1    pencils
dtype: object
```

```python
materials[[1, 0, 2]] # For specific values and order
```

```
1    pencils
0       pens
2      paper
dtype: object
```

Create a Series from a dictionary-keys become indexes, values become Series values:

```python
new_series = pd.Series(fruits_dict)
new_series
```

```
orange              10
lemon     [40, 60, 80]
dtype: object
```

For numerical Series, use `NumPy` for mathematical operations. `NumPy` is a `Python` library for numerical computing:

```python
import numpy as np

numerical_series = pd.Series([0.30, 0.33, 0.36, 0.39])
np.exp(numerical_series) # For exponential
```

```
0    1.349859
1    1.390968
2    1.433329
3    1.476981
dtype: float64
```

```python
numerical_series * 2 # For multiplication
```

```
0    0.60
1    0.66
2    0.72
3    0.78
dtype: float64
```

Note that indexes are preserved.
Logical operations work on numerical Series:

```python
numerical_series > 0.35
```

```
0    False
1    False
2     True
3     True
dtype: bool
```

And also for non-numerical Series:

```
'calculator' in materials
```

```
False
```

A DataFrame is a rectangular, tabular data structure consisting of rows and columns (essentially a collection of Series). DataFrames store data in digital format, similar to spreadsheet tables. DataFrames can handle large amounts of information (billions of items, depending on computer memory). Data can be numeric, string, Boolean, etc. Each cell has an address identified by its row and column. DataFrames have indexes for rows and columns.

We'll create a dictionary abc with information about municipalities in Greater ABC Paulista, São Paulo, Brazil, from the 2022 Demographic Census. The first column includes the names of the municipalities; the second column contains the population; the third is the average salary in terms of minimum wage; the next is the number of workers; and finally, there is a column with the area of the municipality in square kilometers.

```
abc = {'Name': ["Santo André","São Bernardo Do Campo","São Caetano Do Sul",         "Diadema","Rio Granc
    'Population': [748919, 810729, 165655, 393237, 44170, 115559, 418261],
    'Salary': [2.8, 3.6, 3.2, 3, 2.4, 2.6, 3],
    'Workers': [242730, 299290, 119715, 102164, 3838, 24844, 77038],
    'Area_km2': [175782, 409532, 15331, 30732, 36341, 98972, 61937]}

df = pd.DataFrame(abc)
df
```

```
                    Name  Population  Salary  Workers  Area_km2
0            Santo André      748919     2.8   242730    175782
1  São Bernardo Do Campo      810729     3.6   299290    409532
2     São Caetano Do Sul      165655     3.2   119715     15331
3                Diadema      393237     3.0   102164     30732
4    Rio Grande Da Serra       44170     2.4     3838     36341
5        Ribeirão Pires       115559     2.6    24844     98972
6                   Mauá      418261     3.0    77038     61937
```

Different from the previous scalars and data structures, in RStudio, you can click on a DataFrame in the Environment (upper right pane tab) to visualize the data as tabular data in a new tab. You can sort and filter values like in a spreadsheet.

DataFrames are rectangular: all columns must have the same length because each row represents an object of interest (here, a municipality). If information is missing for some rows, we must either exclude those rows or mark missing values appropriately.

There's one exception: when creating a DataFrame from a dictionary with different value lengths, pandas creates rows equal to the maximum length and fills missing values appropriately. For example, using fruits_dict (with 2 keys: 'orange' has 1 value, 'lemon' has 3 values):

```
fruits_dict
```

```
{'orange': 10, 'lemon': [40, 60, 80]}
```

```
pd.DataFrame(fruits_dict)
```

```
   orange  lemon
0      10     40
1      10     60
2      10     80
```

For more than two columns, the table must be rectangular. *You cannot create a DataFrame from a dictionary with keys of different lengths.*

Returning to our municipalities DataFrame, view the first five rows with the method `head()`:

```
df.head()
```

```
                    Name  Population  Salary  Workers  Area_km2
0            Santo André      748919     2.8   242730    175782
1  São Bernardo Do Campo      810729     3.6   299290    409532
2      São Caetano Do Sul     165655     3.2   119715     15331
3                Diadema      393237     3.0   102164     30732
4    Rio Grande Da Serra       44170     2.4     3838     36341
```

Get general information with `.info()`, which shows index type, column names, data types, non-null counts, and memory usage:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Name        7 non-null      object
 1   Population  7 non-null      int64
 2   Salary      7 non-null      float64
 3   Workers     7 non-null      int64
 4   Area_km2    7 non-null      int64
dtypes: float64(1), int64(3), object(1)
memory usage: 408.0+ bytes
```

You can obtain a statistical summary by applying the method `.describe()`. For numeric columns, the describe output contains the count (number of non-null observations), mean, standard deviation, minimum value, percentile values and maximum value.

```
df.describe()
```

```
          Population    Salary        Workers       Area_km2
count       7.000000  7.000000       7.000000       7.000000
mean   385218.571429  2.942857  124231.285714  118375.285714
std    303241.322570  0.395209  109388.026991  139438.531912
min     44170.000000  2.400000    3838.000000   15331.000000
25%    140607.000000  2.700000   50941.000000   33536.500000
50%    393237.000000  3.000000  102164.000000   61937.000000
75%    583590.000000  3.100000  181222.500000  137377.000000
max    810729.000000  3.600000  299290.000000  409532.000000
```

By default, describe() analyzes only numeric columns. To include string/categorical columns, use `include = 'object'` or `include = 'all'`. For object data, the summary includes count, unique values, most frequent value (top), and its frequency:

```
df.describe(include='all')
```

```
               Name     Population    Salary        Workers       Area_km2
count             7       7.000000  7.000000       7.000000       7.000000
unique            7            NaN       NaN            NaN            NaN
top     Santo André            NaN       NaN            NaN            NaN
freq              1            NaN       NaN            NaN            NaN
mean            NaN  385218.571429  2.942857  124231.285714  118375.285714
std             NaN  303241.322570  0.395209  109388.026991  139438.531912
min             NaN   44170.000000  2.400000    3838.000000   15331.000000
25%             NaN  140607.000000  2.700000   50941.000000   33536.500000
50%             NaN  393237.000000  3.000000  102164.000000   61937.000000
75%             NaN  583590.000000  3.100000  181222.500000  137377.000000
max             NaN  810729.000000  3.600000  299290.000000  409532.000000
```

Access DataFrame columns by name within square brackets:

```
df['Name']
```

```
0              Santo André
1    São Bernardo Do Campo
2       São Caetano Do Sul
3                  Diadema
4      Rio Grande Da Serra
5          Ribeirão Pires
6                     Mauá
Name: Name, dtype: object
```

Or using dot notation (attribute access, not recommended if column names contain spaces or conflict with DataFrame methods):

```
df.Name
```

```
0              Santo André
1    São Bernardo Do Campo
2       São Caetano Do Sul
3                  Diadema
4      Rio Grande Da Serra
5          Ribeirão Pires
6                     Mauá
Name: Name, dtype: object
```

And for more than one column:

```
df[['Name','Area_km2']]
```

```
                    Name  Area_km2
0            Santo André    175782
1  São Bernardo Do Campo    409532
2     São Caetano Do Sul     15331
3                Diadema     30732
4    Rio Grande Da Serra     36341
5        Ribeirão Pires     98972
6                   Mauá     61937
```

To access a specific cell:

```
df['Name'][5] # For the row of index 5
```

```
'Ribeirão Pires'
```

To access an entire row, use `.loc`:

```
df.loc[5] # For the row of index 5
```

```
Name          Ribeirão Pires
Population            115559
Salary                   2.6
Workers                24844
Area_km2               98972
Name: 5, dtype: object
```

We can create a column or update values in a column that already exists by doing:

```
df["Test"] = 10
df
```

```
                     Name  Population  Salary  Workers  Area_km2  Test
0            Santo André      748919     2.8   242730    175782    10
1  São Bernardo Do Campo      810729     3.6   299290    409532    10
2     São Caetano Do Sul      165655     3.2   119715     15331    10
3                Diadema      393237     3.0   102164     30732    10
4      Rio Grande Da Serra     44170     2.4     3838     36341    10
5         Ribeirão Pires      115559     2.6    24844     98972    10
6                   Mauá      418261     3.0    77038     61937    10
```

In the case above, a new column "Test" is created with all values set to 10.

Add new rows using `concat()`. In this example, the new dictionary only has values for the "Name" column, so other columns are filled with `NaN`:

```
new_rows = [{"Name": "Test"}]

df = pd.concat([df, pd.DataFrame(new_rows)], ignore_index=True)

df
```

```
                     Name  Population  Salary   Workers  Area_km2  Test
0            Santo André    748919.0     2.8  242730.0  175782.0  10.0
1  São Bernardo Do Campo    810729.0     3.6  299290.0  409532.0  10.0
2     São Caetano Do Sul    165655.0     3.2  119715.0   15331.0  10.0
3                Diadema    393237.0     3.0  102164.0   30732.0  10.0
4      Rio Grande Da Serra   44170.0     2.4    3838.0   36341.0  10.0
5         Ribeirão Pires    115559.0     2.6   24844.0   98972.0  10.0
6                   Mauá    418261.0     3.0   77038.0   61937.0  10.0
7                   Test         NaN     NaN       NaN       NaN   NaN
```

Remove a column with `drop()` and `axis = 1`:

```
df = df.drop('Test', axis = 1)
df
```

```
                    Name  Population  Salary   Workers   Area_km2
0            Santo André    748919.0     2.8  242730.0  175782.0
1  São Bernardo Do Campo    810729.0     3.6  299290.0  409532.0
2     São Caetano Do Sul    165655.0     3.2  119715.0   15331.0
3                Diadema    393237.0     3.0  102164.0   30732.0
4    Rio Grande Da Serra     44170.0     2.4    3838.0   36341.0
5         Ribeirão Pires    115559.0     2.6   24844.0   98972.0
6                   Mauá    418261.0     3.0   77038.0   61937.0
7                   Test         NaN     NaN       NaN       NaN
```

Remove a row with `axis = 0`:

```
df = df.drop(7, axis = 0)
df
```

```
                    Name  Population  Salary   Workers   Area_km2
0            Santo André    748919.0     2.8  242730.0  175782.0
1  São Bernardo Do Campo    810729.0     3.6  299290.0  409532.0
2     São Caetano Do Sul    165655.0     3.2  119715.0   15331.0
3                Diadema    393237.0     3.0  102164.0   30732.0
4    Rio Grande Da Serra     44170.0     2.4    3838.0   36341.0
5         Ribeirão Pires    115559.0     2.6   24844.0   98972.0
6                   Mauá    418261.0     3.0   77038.0   61937.0
```

More Basic Operations

`Python` supports various operations: arithmetic (sums, multiplications), logical (returning True/False), and more.

To perform operations effectively, understand how DataFrames locate information. Each cell has an address (index) that can be referenced in several ways. Two primary methods are `loc` and `iloc`:

- `loc` (label-based): Uses names/labels (inclusive of slice ends).
- `iloc` (integer-based): Uses numerical positions (0-indexed, exclusive of slice ends).

Select the first row of the "Name" column:

```
df.loc[0, 'Name']
```

```
'Santo André'
```

With loc, the first value is the row, the second is the column (same for iloc). Select multiple rows and columns by label:

```
df.loc[0:2, 'Name':'Salary']
```

```
                    Name  Population  Salary
0            Santo André    748919.0     2.8
1  São Bernardo Do Campo    810729.0     3.6
2     São Caetano Do Sul    165655.0     3.2
```

Select specific columns and rows by label:

```
df.loc[[0,2],['Name','Salary']]
```

```
                 Name  Salary
0         Santo André     2.8
2  São Caetano Do Sul     3.2
```

Now using iloc with integer positions. Select the cell at position (0,0):

```
df.iloc[0,0]
```

```
'Santo André'
```

Select multiple rows and columns by position:

```
df.iloc[0:2,0:3]
```

```
                    Name  Population  Salary
0            Santo André    748919.0     2.8
1  São Bernardo Do Campo    810729.0     3.6
```

Select specific columns and rows by position:

```
df.iloc[[0,2],[0,3]]
```

```
                 Name   Workers
0         Santo André  242730.0
2  São Caetano Do Sul  119715.0
```

Note: With iloc, slicing is exclusive of the end position. Think of loc as searching by name (label) and iloc as searching by numbered position. Asking for `df["Name"][1]` is equivalent to `df.loc[1, "Name"]`.

Indexing facilitates operations. For example, calculate the total workers in Santo André and São Bernardo Do Campo:

```
df['Workers'][1] + df['Workers'][2]
```

```
419005.0
```

An issue with with indexing cells this way is that, if other municipality were added to the table, the row numbers of each municipality might change, and as a consequence you might not be referencing the same municipality with those numbers. Another possibility is if the table is very long, you might not even know which municipality is in which row to begin with.

So a better way to index the cells in a DataFrame is by using logical operators, like in the following chunk of code. Here, we are essentially asking for "Workers of (municipality name which is Santo André)" + "Workers of (municipality name which is São Bernardo Do Campo)":

First, let's select the row in which the name of the municipality is equal to "Santo André":

```
df.loc[df['Name'] == "Santo André"]
```

```
          Name  Population  Salary   Workers  Area_km2
0  Santo André    748919.0     2.8  242730.0  175782.0
```

The text inside the square bracket tells Pandas to look at the row with that municipality's names (we index by the name of the municipality, instead of the row number).

Now, let's select only the column that refers to the number of workers:

```
df.loc[df['Name'] == "Santo André", 'Workers']
```

```
0    242730.0
Name: Workers, dtype: float64
```

Text text after the comma asks to look the value in the `Workers` column for the colum label indexed. This output show all the Workers values in which "Name" is equal to Santo Andre. Note that the output also have a index (for cases in which the query result in more than one output). We can select the value by its index:

```
df.loc[df['Name'] == "Santo André", 'Workers'].values[0]
```

```
242730.0
```

Then, we can do the same for same for São Bernardo Do Campo and create a sum operation to obtain total number of workers in São Bernardo Do Campo and in Santo André:

```
df.loc[df['Name'] == "Santo André", 'Workers'].values[0] + df.loc[df['Name'] == "São Bernardo Do Campo"
```

```
542020.0
```

Suppose that you wanted to calculate the area (in square kilometers) in this DataFrame. To do this, you would use `.sum()` in the column selection that you want to sum:

```
df['Area_km2'].sum()
```

```
828627.0
```

As you can see, `Python` can be used as a calculator, but it is much more powerful than that.

You have already seen how `Python` allows you to store in memory the results of some instruction, by means of an assignment `=`. You can also perform many other useful operations. For instance, you can find the maximum for a set of values:

```
df['Area_km2'].max()
```

```
409532.0
```

This does not have to be just the maximum of a column. You can ask for the max of any set of values:

```
df.loc[df['Name'].isin(["Diadema", "Mauá"]), 'Area_km2'].max()
```

```
61937.0
```

And, if you wanted to find the name of the municipality with the largest area, you can do this:

```
df.loc[df['Area_km2'] == df['Area_km2'].max(), 'Name']
```

```
1    São Bernardo Do Campo
Name: Name, dtype: object
```

As you see, São Bernardo Do Campo is the municipality with the largest territorial area. Likewise, the function for finding the minimum value for a set of values is `min`:

```
df['Area_km2'].min()
```

```
15331.0
```

So which of the municipality in the DataFrame had the smallest territorial area?

```
df.loc[df['Area_km2'] == df['Area_km2'].min(), 'Name']
```

```
2    São Caetano Do Sul
Name: Name, dtype: object
```

The data can be explored in fairly sophisticated ways by using indexing in imaginative ways.

Try calculating the mean territorial area using the command `.mean()`. To do this, create a new chunk of code and type your code. The keyboard shortcut to insert code chunks into Quarto Markdown files is CTRL-ALT-I.

A powerful feature of `Python` is the flexibility to use calculations to explore and analyze data. In addition to operations involving a single column in a DataFrame, we can ask `Python` to do calculations using several columns. The sample DataFrame contains information about municipalities in the Grande ABC Paulista. Suppose that you would like to discover which municipality on average has the largest fraction of employed population.

We will define the fraction of employed population by municipality as the number of workers per total population in each municipality. This is calculated as:

$$FE_i = \frac{workers_i}{population_i}$$

Where FE_i is the fraction of employed population in municipality i.

Here we introduce another feature of Quarto Markdown documents: the text above between the `$$` signs is a piece of LaTex code. It allows you to type mathematical formulas in an R Markdown document. Mathematical expressions can be written inline by using the notation x. Do not worry too much about how to write mathematical expressions at the moment, this is something that you can learn when and if needed, and there are many resources online to help you get started.

The fraction of employed population is calculated as follows (the operations are done on a row-by-row basis, that is, the workers in row i are divided by the population in row i):

```
FE = df['Workers']/df['Population']
FE
```

```
0    0.324107
1    0.369162
2    0.722677
3    0.259803
4    0.086892
5    0.214990
6    0.184186
dtype: float64
```

The chunk of code above created a new Series called `FE`. If you wanted to add this vector to the DataFrame as a new column, you could do this instead:

```
df["FE"] = df['Workers']/df['Population']
```

As you can see, it is possible to create new columns on the fly and assign stuff to them, as long as the size of what we are assigning is compatible with the DataFrame (i.e., same number of rows). You can check that the new column was added to your existing `df` DataFrame:

```
df
```

```
                     Name  Population  Salary   Workers  Area_km2        FE
0            Santo André    748919.0     2.8  242730.0  175782.0  0.324107
1  São Bernardo Do Campo    810729.0     3.6  299290.0  409532.0  0.369162
2     São Caetano Do Sul    165655.0     3.2  119715.0   15331.0  0.722677
3                Diadema    393237.0     3.0  102164.0   30732.0  0.259803
4    Rio Grande Da Serra     44170.0     2.4    3838.0   36341.0  0.086892
5         Ribeirão Pires    115559.0     2.6   24844.0   98972.0  0.214990
6                   Mauá    418261.0     3.0   77038.0   61937.0  0.184186
```

If you would like to round off numeric data values, you could use the `round()` function Here, we round to two decimals:

```
df['FE'] = df['FE'].round(2)
df
```

```
                     Name  Population  Salary   Workers  Area_km2    FE
0            Santo André    748919.0     2.8  242730.0  175782.0  0.32
1  São Bernardo Do Campo    810729.0     3.6  299290.0  409532.0  0.37
2     São Caetano Do Sul    165655.0     3.2  119715.0   15331.0  0.72
3                Diadema    393237.0     3.0  102164.0   30732.0  0.26
4    Rio Grande Da Serra     44170.0     2.4    3838.0   36341.0  0.09
5         Ribeirão Pires    115559.0     2.6   24844.0   98972.0  0.21
6                   Mauá    418261.0     3.0   77038.0   61937.0  0.18
```

Knitting

Knitting is the process of converting the Markdown document (source) into some other type of document (output). The output could be HTML, a Word file, or a PDF file. If you check the header of this document, you will see that it is configured to knit into a PDF file, with certain parameters, for example, using the file `exercise-template-default.tex` to convert to LaTeX and hence to PDF. In the background, knitting uses Pandoc to convert between document formats.

Before rendering this document, make sure that you have installed {tinytex} (see the Quick Start Guide of the {edashop} package). {tinytex} is a lightweight LaTeX distribution:

```
tinytex:::is_tinytex()
```

```
[1] TRUE
```

Practice

1. Describe in your own words the concept of literate programming. What is a computational notebook?
2. Load the São Paulo Social Vulnerability Index (IVSP) dataset at census tract level (included in the package {edashop}). To make it accessible in Python, we first load it in R and then access it via reticulate.

Loading the R dataset:

```
library(edashop)
library(reticulate)

data("cntr_sp_ipvs",
     package = "edashop")
```

Transforming it into a Pandas DataFrame:

```
IVSP = r.cntr_sp_ipvs
type(IVSP)
```

```
<class 'pandas.core.frame.DataFrame'>
```

Visualizing the first five rows:

```
IVSP.head()
```

```
         COD_SETOR          AGSN  ... name_district  code_state
0  350010505000001  Não especial  ...    Adamantina          35
1  350010505000002  Não especial  ...    Adamantina          35
2  350010505000003  Não especial  ...    Adamantina          35
3  350010505000004  Não especial  ...    Adamantina          35
4  350010505000005  Não especial  ...    Adamantina          35

[5 rows x 29 columns]
```

3. How many variables are there in the data frame, and what data types are they (i.e., numerical, logical, object, etc.)? (Hint: check `.info()`)

4. What is the maximum average income of the head of the household? In which municipality is this value found? (Hint: check `??cntr_sp_ipvs`)

```
??cntr_sp_ipvs
```

```
starting httpd help server ... done
```

5. Which municipality has the census tract with the highest number of permanent private domiciles?

6. Calculate the mean of proportion of domiciles with no per capita income.

7. Calculate the fraction of the improvised private domiciles relative to the total number of private and collective domiciles.

8. What is the sum of permanent private domiciles in the municipalities of Diadema, Ribeirão Preto and São Paulo?