Lab 2: Data Classes and Programming Fundamentals

PSTAT 5A - Summer Session A 2025

Instructor: Narjes Mathlouthi

2025-06-30

Table of contents



## 1 Introduction

Last week, we were introduced to the notion of data types. Recall that “data type” can be thought of as the category (or type) of data i.e. integer, float, character, etc.

In Python, however, we often need to aggregate data into larger structures, often referred to as **data classes**.

## 2 Lists

Perhaps the most fundamental data structure in Python is that of a **list**. Just like lists in real life or in mathematics, Python lists are just collections of items enclosed in square brackets:

[<item 1>, <item 2>, ..., <item n>]

Again, the items in a list can be of any data type; we can even mix and match data types!

### 2.1 Task 1

Create a list containing the elements 1, "hi", 3.4, and "PSTAT 5A". Assign this list to a variable called list1.

# Your code here

Just as we were able to use a Python function (type()) to check the type of a particular piece of data, we can also use Python to check the structure or class of a piece of data. It turns out that we use the same function as before- namely, type()!

### 2.2 Task 1 (cont’d)

Run the code type(list1).

# Your code here

## 3 Indexing

Alright, now that we can store data in lists, how can we access elements in a list? The answer is to use what is known as **indexing**.

Given a list x, we access the ith element using the code:

x[i]

The reason we call this “indexing” is because the number that goes between the brackets is the **index** of the element that we want.

|  |
| --- |
| Caution |
| Python begins indexing at 0. |

What does this mean? Well, let’s see by way of an example.

### 3.1 Task 2

Create a list with the numbers 1 through 10, inclusive, and assign this to a variable called x.

# Your code here

Run the code x[1].

# Your code here

Run the code x[0].

# Your code here

So, what we would colloquially call the first element of a list, Python calls the **zeroeth** element.

### 3.2 Task 3

Create a list called x that contains the elements 1, "two", 3.5, "four", and "five five". Answer the following questions **WITHOUT** running any code, writing your answers as a comment in a code cell:

1. What would be the output of type(x)?
2. What would be the output of type(x[1])?
3. What would be the output of x[0]?

# Create the list  
x = [1, "two", 3.5, "four", "five five"]  
  
# Your predictions as comments:  
# 1. type(x) would output:   
# 2. type(x[1]) would output:   
# 3. x[0] would output:

Now, run code to verify your answers to the above three questions.

# Verify your answers here

## 4 Tables

Another very useful data structure in Python is that of a **table**. Python tables behave pretty much the same as the tables we’ve used in, say, math- they are a grid of values arranged sequentially.

Tables can be created using the Table() function in Python, which itself comes from the datascience module. The general syntax of creating a table with the Table() function is:

Table().with\_columns(  
 "", [, , ... ],  
 "", [, , ... ],  
 ...  
)

For example:

# Install datascience if needed (uncomment if necessary)  
# !pip install datascience  
from datascience import \*  
  
Table().with\_columns(  
 "Course", ["PSTAT 5A", "PSTAT 120A", "PSTAT 130"],  
 "Units", [4, 4, 4],  
 "Quarter", ["Summer", "Fall", "Winter"]  
)

| Course | Units | Quarter |
| --- | --- | --- |
| PSTAT 5A | 4 | Summer |
| PSTAT 120A | 4 | Fall |
| PSTAT 130 | 4 | Winter |

There is nothing stopping us from assigning a table to a variable! For example, after running:

table1 = Table().with\_columns(  
 "Course", ["PSTAT 5A", "PSTAT 120A", "PSTAT 130"],  
 "Units", [4, 4, 4],  
 "Instructor", ["Mathlouthi", "Johnson", "Smith"]  
)  
  
table1

| Course | Units | Instructor |
| --- | --- | --- |
| PSTAT 5A | 4 | Mathlouthi |
| PSTAT 120A | 4 | Johnson |
| PSTAT 130 | 4 | Smith |

### 4.1 Terminology

Sometimes in Python we will encounter expressions of the form:

<object type>.<function name>()

In this syntax, the function <function name> is said to be a **method**. For example, the function with\_columns() is a method for the Table object.

The datascience module contains a plethora of methods we can use to manage tables. For example, the select() method can be used to select columns by name:

table1.select("Units")

| Units |
| --- |
| 4 |
| 4 |
| 4 |

|  |
| --- |
| Syntax |
| Methods are always appended to either a function that creates a blank object type (like Table()) or a variable of the correct type. |

### 4.2 Task 4

Read the list of methods for Table objects at <http://data8.org/datascience/tables.html>, and write down (in a code cell, using comments) at least three different methods, including a short description of what each method does. For example:

# .with\_columns(): adds specified columns to a table.  
# Your three methods here:

### 4.3 Task 5

Create the following table, and assign it to a variable called profs:

| Professor | Office | Course |
| --- | --- | --- |
| Dr. Swenson | South Hall | PSTAT 130 |
| Dr. Wainwright | Old Gym | PSTAT 120A |
| Dr. Mouti | Old Gym | PSTAT 126 |

# Your code here

Run a cell containing only the code profs to make sure (visually) that your table looks correct.

# Your code here

Select the column called Course from profs.

# Your code here

Create a new table called profs\_new that contains the same rows as the profs table, but with the following additional row:

| Professor | Office | Course |
| --- | --- | --- |
| Dr. Ravat | South Hall | PSTAT 120B |

# Your code here

Run a cell containing only the code profs\_new to make sure (visually) that the appending was successful.

|  |
| --- |
| Tip |
| Think about how you can use our discussion on updating variable values from last lab. Also, the method .with\_row() may be useful; see the help file at <http://data8.org/datascience/tables.html> for more information. |

# Your code here

### 4.4 Filtering Tables

Suppose we want to select rows of a table that satisfy a given condition. For example, if we wanted to find the information of only courses taught by Mathlouthi in the table1 table above, we would call:

table1.where("Instructor", "Mathlouthi")

| Course | Units | Instructor |
| --- | --- | --- |
| PSTAT 5A | 4 | Mathlouthi |

What would happen if we tried to select the rows of table1 with “Wilson” in the Instructor column? Well, since there is nobody in table1 named Wilson, we should hope that Python returns an empty table.

table1.where("Instructor", "Wilson")

| Course | Units | Instructor |
| --- | --- | --- |

Sure enough, Python has returned an empty table!

## 5 Arrays

The final Data Structure we will examine in this class is that of an **array**. Arrays behave very similarly to Tables, with a few differences. For one, the syntax used to create an array is slightly different:

make\_array(<item 1>, <item 2>, <item 3>, ...)

For example:

make\_array("Spring", "Summer", "Autumn", "Winter")

array(['Spring', 'Summer', 'Autumn', 'Winter'],  
 dtype='<U6')

You may ask- what’s that dtype='<U6' symbol at the end of the output? For now, don’t worry about it, as we will revisit this later.

### 5.1 Lists vs. Arrays

So, we now know about three different data classes in Python: lists, tables, and arrays. At first glance, lists and arrays may seem somewhat similar. However, there are a few key differences between them:

### 5.2 Task 6

Make a list called my\_list containing the elements 1, 2, and 3, and make an array called my\_array also containing the elements 1, 2, and 3. Run the following commands in separate code cells:

# Create list and array  
my\_list = [1, 2, 3]  
my\_array = make\_array(1, 2, 3)

sum(my\_list)

6

sum(my\_array)

6

# my\_list + 2 # This will cause an error

my\_array + 2

array([3, 4, 5])

What the previous Task illustrates is the fact that **arrays lend themselves to element-wise operations**, whereas lists do not. One important limitation about arrays, though, is that the elements in an array must all be of the same data type. If you try to make an array consisting of elements that are different data types Python will still run, however it will not run in the way you expect it to!

## 6 Comparisons

Here’s a question: is 2 less than 3? Well, yes it is! If we wanted to confirm this, we could simply ask Python whether 2 is less than 3 by running:

2 < 3

True

Notice, however, how Python answered this question: it simply returned True. Let’s see what the data type of True is:

type(True)

bool

True is of the type bool, which is short for **boolean**. There are only two boolean quantities in Python: True and False. Let’s see how we can generate a False value:

3 < 2

False

Here is a list of comparison operators, taken from the Inferential Thinking textbook:

| Comparison | Operator | True Example | False Example |
| --- | --- | --- | --- |
| Less than | < | 2 < 3 | 2 < 2 |
| Greater than | > | 3 > 2 | 3 > 3 |
| Less than or equal | <= | 2 <= 2 | 3 <= 2 |
| Greater than or equal | >= | 3 >= 3 | 2 >= 3 |
| Equal | == | 3 == 3 | 3 == 2 |
| Not equal | != | 3 != 2 | 2 != 2 |

One nice thing about Python is that it allows for multiple simultaneous comparisons. For example:

2 < 3 < 4

True

|  |
| --- |
| Important |
| In a multiple comparison, Python will only return True when **all** of the included comparisons are true. |

For instance, 2 < 3 < 1 would return False, because even though 2 is less than 3 it is not true that 3 is less than 1.

Believe it or not, you can compare strings as well! Python compares strings alphabetically; that is, letters at the beginning of the alphabet are considered to have smaller ordinal value than letters at the end of the alphabet. For example:

"apple" < "banana"

True

"zebra" < "zanzibar"

False

"cat" <= "catenary"

True

### 6.1 Task 7

Check how "statistics" and "Statistics" (note the capitalization!) compare. Use this to answer the question: when Python is comparing strings, does it give precedence to capital letters or not? If so, which (lowercase or capital) is given a “higher” value?

# Your code here

### 6.2 Comparing Lists and Arrays

Finally, we discuss how comparisons work in the context of lists and arrays. The way Python compares lists is by what is known as **lexicographical order**. From the official Python help documentation, this means:

first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted.

For instance, [1, 2, 3] < [2, 1, 1] would return True since 1 (the first element of the first list) is less than 2 (the first element of the second list).

The comparison of arrays is a little more straightforward, except:

|  |
| --- |
| Important |
| When comparing two arrays, the arrays must be of the same length. |

To see exactly how comparison of arrays works, let’s work through a Task:

### 6.3 Task 8

Make an array with the elements 1, 2, and 3, and call this x. Make another array with the elements 2, 3, 1, and call this y. Run x < y, and comment on the result.

# Your code here

What the previous task illustrates is that Python compares arrays **element-wise**.

## 7 Conditionals

Now, we can use comparisons for much more than verifying simple arithmetic relationships. One of the main areas in which comparisons arise is the area of **conditional expressions**.

Simply put, conditional expressions are how we can convey a set of choices to Python. As an example, let’s consider finding someone’s city based on their zip code. To simplify, let’s assume the only zip codes we consider are 93117, 93120, and 93150. From postal data, we know that:

* a zip code of 93117 corresponds to Goleta
* a zip code of 93120 corresponds to Santa Barbara
* a zip code of 93150 corresponds to Montecito

We can rephrase this information in terms of “if” statements:

* **If** a person has a zip code of 93117, **then** they are in Goleta
* **Otherwise, if** they have a zip code of 93120, **then** they are in Santa Barbara
* **Otherwise, if** they have a zip code of 93150, **then** they are in Montecito

This is precisely the syntax we would use when translating this experiment into Python syntax:

# Example of conditional statements with zip codes  
# First, we need to define a zip\_code variable  
zip\_code = 93117 # Example zip code  
  
if zip\_code == 93117:  
 location = "Goleta"  
elif zip\_code == 93120:  
 location = "Santa Barbara"  
elif zip\_code == 93150:  
 location = "Montecito"  
else:  
 location = "Unknown location"  
  
print(f"Zip code {zip\_code} corresponds to: {location}")  
  
# Test with different zip codes  
for test\_zip in [93117, 93120, 93150, 99999]:  
 if test\_zip == 93117:  
 loc = "Goleta"  
 elif test\_zip == 93120:  
 loc = "Santa Barbara"  
 elif test\_zip == 93150:  
 loc = "Montecito"  
 else:  
 loc = "Unknown location"  
 print(f"Zip code {test\_zip}: {loc}")

Zip code 93117 corresponds to: Goleta  
Zip code 93117: Goleta  
Zip code 93120: Santa Barbara  
Zip code 93150: Montecito  
Zip code 99999: Unknown location

By the way: elif is an abbreviation for **else if**, which itself can be thought of as equivalent to **otherwise, if**.

Here’s the general syntax of a conditional expression in Python:

if <condition 1>:  
 <task 1>  
elif <condition 2>:  
 <task 2>  
...  
else:  
 <final task>

When executing the above conditional statement, Python first checks whether <condition 1> returns a value of True or False. If it returns a value of True, then <task 1> is executed and the statement ends. Otherwise, Python checks whether <condition 2> is True or False; if it is True then <task 2> is executed, etc.

|  |
| --- |
| Important |
| In the example code above: if <condition 1> is True, then no tasks beyond <task 1> are evaluated. If <condition 2> is True, then no tasks beyond <task 2> are evaluated. And so on and so forth. |

### 7.1 Task 9

Consider the code:

x = 2  
  
if x < 2:  
 x = "hello"  
elif x < 3:  
 x = "goodbye"  
else:  
 x = "take care"

Before running any code, write down what you think the result of executing x would be. Then, run the loop, execute x, and check whether your answer was correct or not.

# Your prediction:   
  
# Now run the code:  
x = 2  
  
if x < 2:  
 x = "hello"  
elif x < 3:  
 x = "goodbye"  
else:  
 x = "take care"  
  
print(x)

goodbye

|  |
| --- |
| Caution |
| Indentation is very important in Python. |

For example, if instead of the conditional expression in Task 9 we had instead put:

x = 2  
  
if x < 2:  
x = "hello"  
elif x < 3:  
x = "goodbye"  
else:  
x = "take care"

then we would have received an error!

## 8 Functions

Finally, let’s quickly discuss Python functions. We’ve already been using quite a few functions:

### 8.1 Task 10

In a Markdown cell, write down three functions we’ve used in this Lab thus far.

**Your answer here:**

If you recall, the general syntax for calling a function is:

<function name>(<arg1>, <arg2>, ... )

where <function name> denotes the function name and <arg1>, <arg2>, etc. denote the arguments of the function.

Creating your own function in Python is actually fairly simple! Here is the syntax we use:

def <function name>(<list out the argument names>):  
 """include a 'docstring' here"""  
 <body of the function>  
 return <what you want the function to output>

For example:

def f(x, y):  
 """returns x^2 + y^2"""  
 return x\*\*2 + y\*\*2

creates a function f that can be called on two arguments, x and y, and returns the sum of squares of the arguments; e.g.

f(3, 4) # should return 3^2 + 4^2 = 25

25

By the way, the **docstring** referenced above is a verbal description of what the function does. (Recall from Lab1 that it is just a multi-line comment, since it is enclosed in triple quotation marks!). All functions should include a docstring to convey to the user what the function does.

|  |
| --- |
| Important |
| If you don’t include a return statement in the definition of a function, then your function will never return anything. |

For instance:

def g(x, y):  
 """should return x^2 + y^2"""  
 x\*\*2 + y\*\*2  
  
print(g(3, 4)) # This will print None!

None

### 8.2 Task 11

Write a function called cent\_to\_far() which takes in a single temperature c as measured in degrees Celsius and returns the corresponding temperature in degrees Fahrenheit. Check that cent\_to\_far(0) correctly returns 32 and cent\_to\_far(20) correctly returns 68.

As a reminder, the conversion formula is: F = (9/5) × C + 32

# Your code here  
def cent\_to\_far(c):  
 """Convert Celsius to Fahrenheit"""  
 # Your implementation here  
 pass  
  
# Test your function  
print(cent\_to\_far(0)) # Should return 32  
print(cent\_to\_far(20)) # Should return 68

Finally, let’s combine some things by way of a concluding Task:

### 8.3 Task 12

Write a function called parity() that returns the parity (i.e. whether a number is even or odd) of an input x. Call your parity() function on 2 and then 3 to make sure your function behaves as expected. Some hints:

* Recall that % is the modulus operator in Python. Specifically, x % y returns the remainder of performing x divided by y.
* Recall that even numbers are divisible by 2 (so what does this mean about the remainder of dividing x by 2 if x is even?)

# Your code here  
def parity(x):  
 """Returns 'even' if x is even, 'odd' if x is odd"""  
 # Your implementation here  
 pass  
  
# Test your function  
print(parity(2)) # Should return 'even'  
print(parity(3)) # Should return 'odd'

## 9 Summary

In this lab, you’ve learned about:

* **Lists**: Ordered collections that can contain mixed data types
* **Indexing**: Accessing elements using zero-based indexing
* **Tables**: Structured data with named columns using the datascience module
* **Arrays**: Homogeneous collections that support element-wise operations
* **Comparisons**: Boolean operations and comparison operators
* **Conditionals**: if/elif/else statements for decision making
* **Functions**: Creating reusable code blocks with def statements

These are fundamental building blocks that will be essential for the rest of the course!