

Foundations of Data Science for Biologists

Introduction to Python

BIO 724D

2025-FEB-25

Instructors: Greg Wray and Paul Magwene

Why learn another programming language?

Why learn Python?

Expand your computing skills

Analyze non-tabular data (e.g., images, sound, text, arrays, etc.)

Carry out modeling and simulations

Use artificial intelligence approaches

Automate tasks and instruments (e.g., use Arduino or Raspberry Pi to capture data)

“Glue” code from multiple languages into a single workflow (bash, C, SQL, R, etc.)

Increase your value to future collaborators, colleagues, mentees, and employers

Useful in academia, industry, consulting, teaching, etc.

What is Python?

General programming language (GPL)

Provides a robust set of built-in data structures optimized for different purposes

Easily adaptable to a wide range of applications: non-tabular data, simulations, etc.

Offers powerful object-oriented programming features

Well-suited for machine learning/artificial intelligence

Emphasis on ease of code development/maintenance over raw performance

Syntax is very clear, more so than just about any other language

Large, versatile standard library (“batteries included” philosophy)

Uses numerous functional programming features to simplify code

Widely considered one of the best languages for collaborative coding

Exception handlers and error reporting are powerful and customizable

Starting out with a new language

The good news: your experience with R will help (a lot!)

Basic programming concepts remain the same

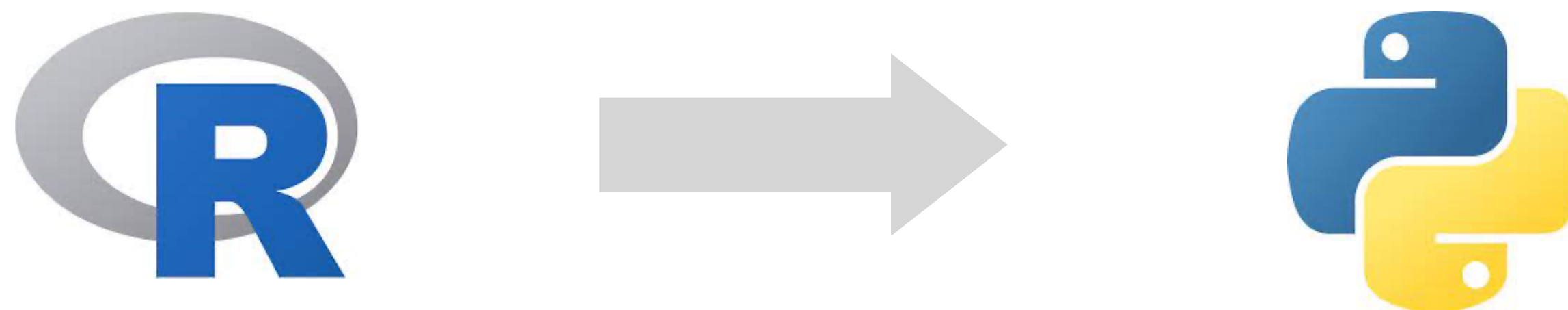
Python is more similar to R than many other languages (e.g., C, Java, Rust)

The challenge: some key differences from R can create unexpected errors and results

Today's goals:

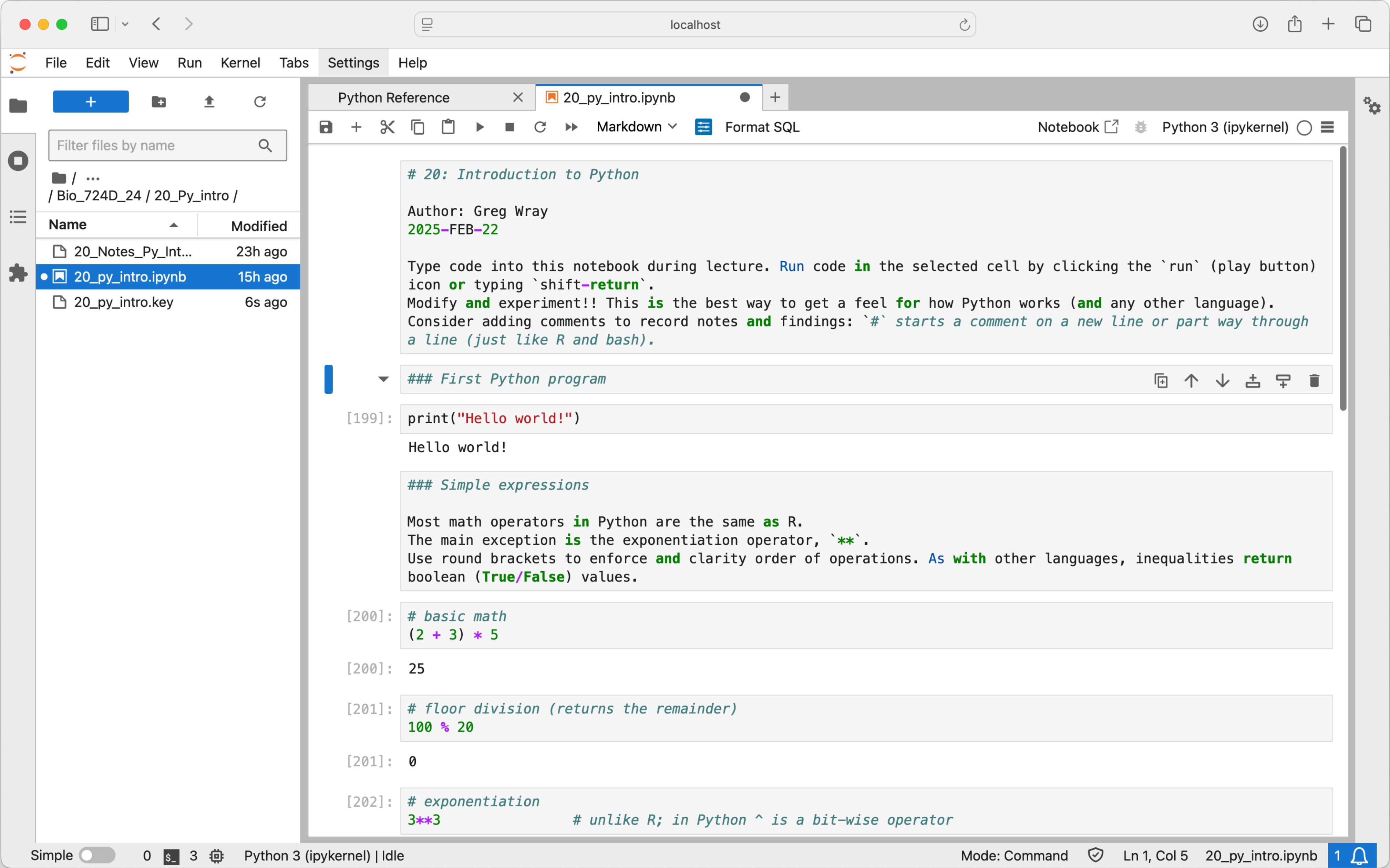
Get you working with Python code as quickly as possible

Minimize the frustration caused by **differences** between the two languages



JupyterLab notebooks

The JupyterLab interface



Quick orientation to the JupyterLab interface

Icons directly above windows provide most basic functions (mouse over for descriptions)

To select a cell: click to its left *or* directly on the gray box

The current cell is indicated by a blue bar at the left

To run code in a cell: select it, then press the play icon or **shift-return**

To edit code in a cell: select it, then click to position the cursor; you can now type

To add, move, or delete a cell: select it, then use the icons at the upper right

To change a cell between code and markdown, select it and use the drop-down menu

Introduction to Python for R programmers

1 Assignment and operators

Basic assignment works similarly to R

Data types are inferred from values (*dynamic typing*)

Variable names can be re-assigned to new values / types (*re-binding*)

Data types can be converted where sensible (called *casting* in Python)

Most of the common **operators** work similarly to R

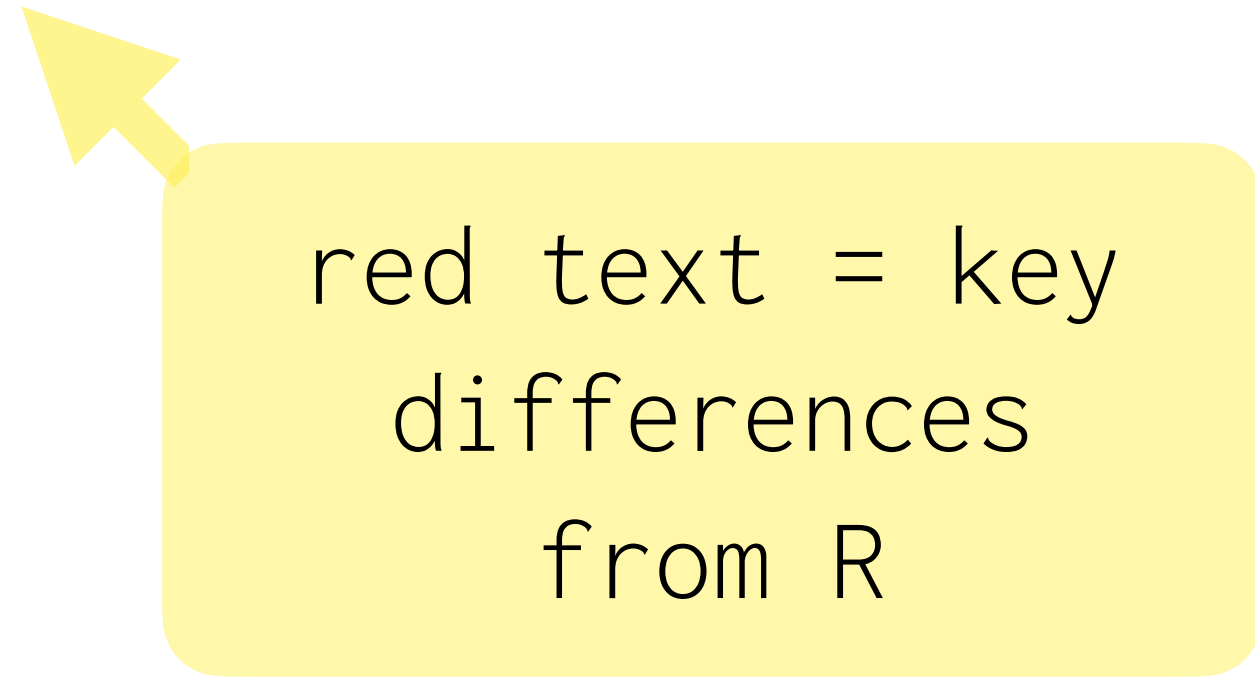
Arithmetic: `+`, `-`, `*`, `/`, `%`, `//` (but use `**` for exponentiation, not `^`)

String: `+` (concatenate)

Comparison: `==`, `!=`, `>`, `<`, `>=`, `<=`

Logical: `and`, `or`, `not`

Membership: `in`, `not in`



red text = key
differences
from R

2 Data types and data structures

Data types: `int`, `float`, `str`, `bool`, `complex`

Similar to R, except that they hold **single values, not vectors of values**

Data structures: `list`, `tuple`, `set`, `frozenset`, `dictionary`

`list` is similar to R: an ordered, general-purpose container

The others have no direct counterparts in base R or the Tidyverse

Many **additional standard data types / structures** are available

Examples: `bytes`, `bytearray`, `date`, `datetime`, `timezone`, `deque`, `heap`

A few commonly used data types are available through third-party libraries

NumPy: `ndarray` (n-dimensional arrays built from many different base data types)

Pandas: `Series` (named list) and `DataFrame` (enhanced data frame)

2.1 Identifying common data types and data structures

Atomic types can be distinguished by their **values** (similar to R atomic vectors):

integer: number with no decimal

float: number with a decimal, even if followed by no decimal values or a zero

bool: `True`, `False` (no quotes)

string: quotes surrounding characters (including only numerals or whitespace)

Data structures can be distinguished by **brackets/braces**:

list: item(s) enclosed in square brackets

`['a', 'b', 'c']`

tuple: item(s) enclosed in round brackets

`('a', 'b', 'c')`

set: item(s) enclosed in curly braces

`{'a', 'b', 'c'}`

dictionary: key:value pair(s) in curly braces

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

Use `type()` to identify the type/class of any data object

3 Iterable data objects

Iterable means that a data object can return one value or item at a time

In Python, **only containers and strings are iterable**

In Python, atomic data types (**int**, **float**, **str**, **bool**, etc.) hold **single values**

Unlike R, where most data types are iterable (atomic types are vectors)

Iteration works slightly differently for containers and strings:

Containers are iterable as a sequence or set of **items**

Strings are iterable as a sequence of **characters** (including whitespaces)

All iterables (except strings) can hold a mix of data types

Unlike R, where only lists can contain items of different types

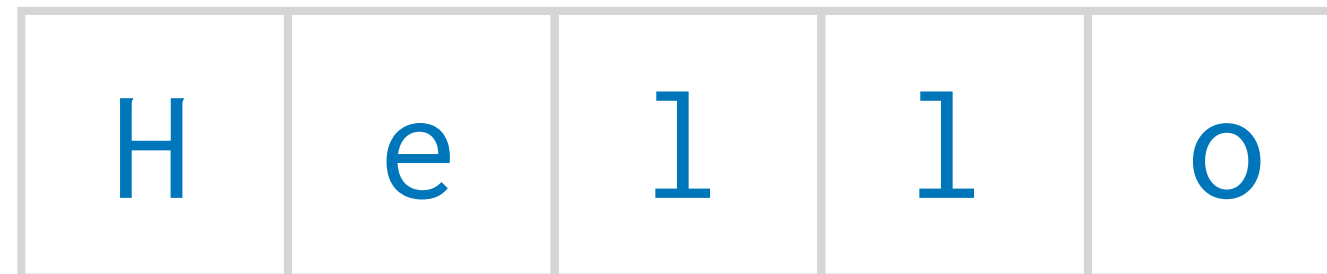
3.1 Strings in Python

Strings are vectors of characters in Python

Strings are indexable and iterable in both Python and R

However, in Python indexing refers to individual characters, not items

```
py_string = 'Hello'
```



```
r_string <- 'Hello'
```



```
py_string[0]
```

returns 'H'

```
r_string[1]
```

returns 'Hello'

4 Indexing

Python uses **square-bracket indexing** for most iterables

This works similarly to indexing in R, with **two differences** (next slide)

Standard dictionaries use **name-based indexing**

Pandas Series and DataFrame data objects can use **either**

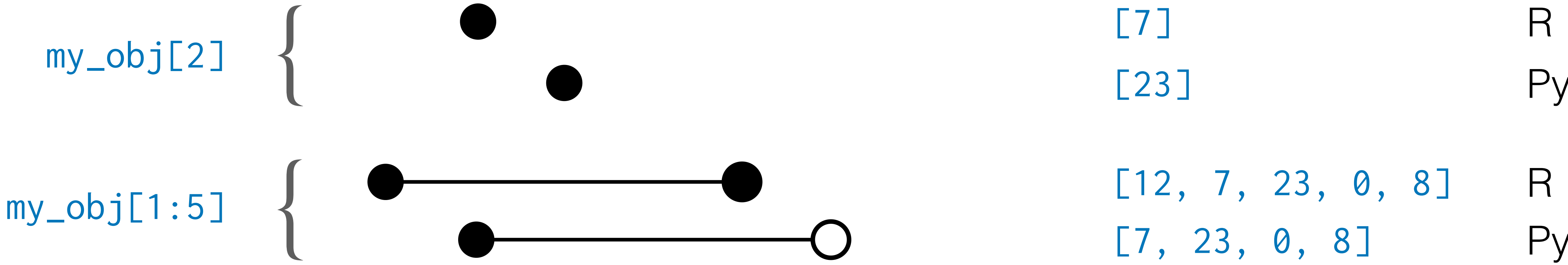
4.1 Square-bracket indexing

Square-bracket indexing in Python is different from R in two ways:

- Zero-based
- Slices do not include the end number

my_obj	12	7	23	0	8	-2	4	2	← values
	1	2	3	4	5	6	7	8	← R index
	0	1	2	3	4	5	6	7	← Python index

Examples:



4.2 Examples of indexing in Python

my_obj	12	7	23	0	8	-2	4	2	← values
	0	1	2	3	4	5	6	7	← index

my_obj[1]	[7]	single item
my_obj[4:5]	[8]	slice of length 1
my_obj[3:]	[0, 8, -2, 4, 2]	open slice to end
my_obj[:2]	[12, 7]	open slice from start
my_obj[:]	[12, 7, 23, 0, 8, -2, 4, 2]	all items
my_obj[-1]	[2]	last item
my_obj[0:7:3]	[12, 0, 4]	slice with step = 3
my_obj[-1:1:-1]	[2, 4, -2, 8, 0, 23]	slice with reverse step
my_obj[::-1]	[2, 4, -2, 8, 0, 23, 7, 12]	reverse (“Martian smiley”)

5 Mutable and immutable objects

Mutable data objects can be modified, **immutable** objects cannot

Immutable data objects are useful as “read only” information

Immutable data objects occupy less memory and provide much faster performance

Immutable objects *can* be deleted

Re-binding the name of an immutable object to a different object deletes the data

Mutability only refers to whether data contained in an object can be changed

R does not provide immutability as a feature

5.1 Mutable and immutable objects, continued

Assigning a mutable data object to a new identifier does not create a copy

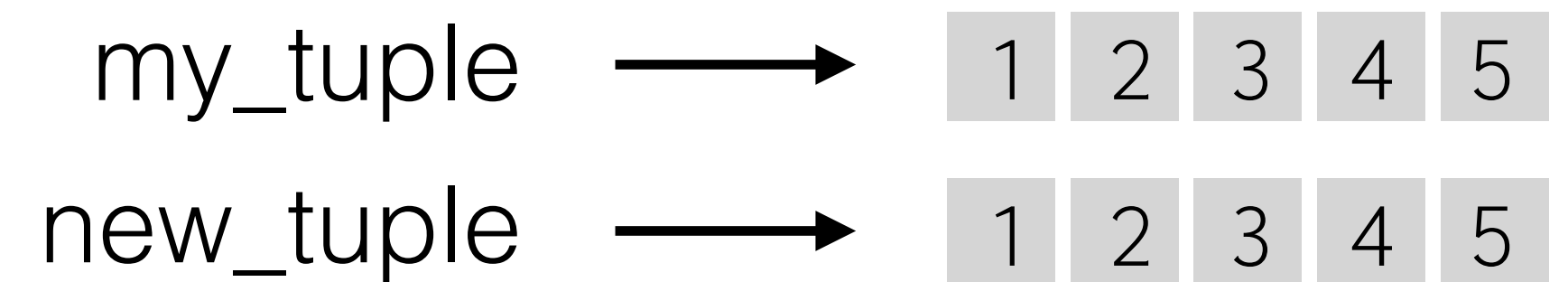
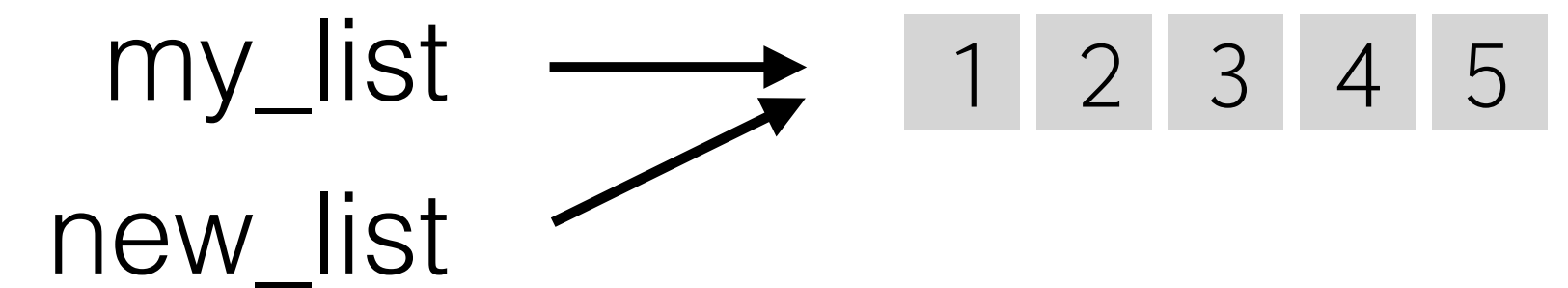
This process creates an alias, but points to the same information in memory

Updating either the original or new identifier changes the data for *both*

This is not true for immutable objects; instead, a true copy is created

```
# assign existing mutable to new identifier
my_list = [1, 2, 3, 4, 5]
new_list = my_list

# assign immutable object to new identifier
my_tuple = (1, 2, 3, 4, 5)
new_tuple = my_tuple
```



5.2 Mutable and immutable objects, continued

Many functions work differently when applied to mutable and immutable objects

Mutable objects are typically altered *in place*

The function itself silently returns **None**

Assignment while applying a mutating function will create an undesirable result!

Any function that alters contents cannot be applied to an immutable object

```
# apply sort to a mutable object
my_list = [3, 1, 5, 2, 4]
my_list.sort()
output = my_list.sort()

# now apply sort to an immutable object
my_tuple = (3, 1, 5, 2, 4)
my_tuple.sort()
```

my_list	→	3	1	5	2	4
my_list	→	1	2	3	4	5
output	→	None				
my_tuple	→	3	1	5	2	4

error!

6 Syntax and formatting

Python was designed to have very clear syntax

Expressions often contain fewer characters than other programming languages

Expressions mimic statements in English as much as possible

Indenting is required to indicate code blocks

Best practice is to use 4 spaces

Any number of tabs or space is allowed, but must be consistent

Indent a second time to indicate a nested code blocks (and so forth; no limit to nesting)

The [Style Guide for Python Code](#) provides detailed advice for clean code

6.1 Control flow

Python provides `if`, `for`, and `while` structures (and some other, less common ones)

The basic syntax is similar to R, although a few additional options are available

Test conditions are expressed in a similar way

To create a control structure:

Use colon and indenting rather than curly braces to define the code block

Stop indenting to indicate the end of the code block

```
# test for life stage of frogs
if gills == True:
    cohort = 'tadpole'
else:
    cohort = 'adult'
print('Cohort is', cohort)
```

```
# print out names of study group
for x in study_group:
    print('name:', x)
num = len(study_group)
print('total individuals =', num)
```


6.2 Functions

Python has two representations of functions

Function: syntax and usage similar to functions in R

Method: function bound to a specific data class; they have a distinct syntax

```
# using a normal function
my_list = [5, 3, 1, 2, 4]          # create a list
function_result = len(my_list)    # call a function
function_result                   # returns 5

# using a method
my_list.sort()                    # methods are attached to data objects
my_list                           # returns [1, 2, 3, 4, 5]
my_string = 'hello world'
my_string.sort()                  # AttributeError (strings are immutable)
```

5.2 Syntax for creating a function

Creating a function in Python uses different syntax from R

Use the `def` keyword rather than calling the `function()` function

Use colon and indenting rather than curly braces to define the code block

Do not use an assignment operator

```
# create a function to test whether 7 is a factor
def is_div_by_7(input_value):
    if input_value % 7 == 0: return True
    else: return False
# call the function
result_1 = is_div_by_7(10)
result_2 = is_div_by_7(42)
print(result_1, result_2)                # returns False True
```

7 Libraries

A vast constellation of useful libraries is available for Python (as for R)

Standard modules: these are built and maintained by the Python Software Organization
Quite extensive and provide many functions and data classes (“batteries included”)
Do not need to be installed: distributed with Python installations and updates

Third-party packages: these are built and maintained by others (like packages in R)
Quality, reliability, and compatibility with updates varies enormously
Some are widely used and extremely reliable (e.g., SciPy, NumPy, Matplotlib, Pandas)
Packages need to be installed before use and manually updated

7.1 Importing libraries

To access a library, you first need to import it within your program (no quotes):

```
import csv
```

Provides access to all data structures and functions in the `csv` module

Best practice: place all import statements at the beginning of a program or module

Multiple libraries can be imported by separating names with commas:

```
import csv, math, time
```

Aliases provide a convenient way to abbreviate names for later use:

```
import numpy as np, pandas as pd
```

7.2 Using functions in libraries

To use an imported function, precede with library name and a period:

```
my_result = factorial(3)      # generates NameError (factorial not defined)
import math                  # factorial() is part of the math module
my_result = factorial(3)     # still generates a NameError
my_result = math.factorial(3) # correct syntax!
my_result                    # now it returns the result: 6
```

This requirement prevents “name collisions” between functions in multiple libraries

Note that R uses a different mechanism: masking function names

Resources for Python

Resources for learning Python

Textbooks

Downey (2015) *Think Python* (2ed). Green Tea Press: [here](#)

McKinney (2022) *Python for Data Analysis* (3ed). O'Reilly: [here](#) (NumPy and Pandas)

Tutorials

Software Carpentry: [here](#)

LinkedIn Learning: login through Duke Libraries

Resources for Python programming

Official Python documentation

The Python Language Reference: [here](#)

The Python Standard Library: [here](#)

General Python programming reference

Martelli et al. (2017) *Python in a Nutshell* (3ed). O'Reilly: [here](#)

Glossaries of Python terminology

Official Python Glossary: [here](#)

Python Lingo from Fluent Python: [here](#)

Python style guide

PEP 8: [here](#)

