Python Handout

Table of Contents

Conda 2
Running Python in the command line 2
Spyder 3
Variables 3
Print 4
Commenting 4
Equal/not equal 5
Types 5
Boolean 6
Lists 6
Indexing 7
Slicing 7
If statements 8
For loops 9
Counting 10
File handling in Python 11
Biopython 12
System arguments 12
While loop 13
Functions 14
Dictionaries 14

Conda

Conda is a package and environment management system. With it, you can load an environment with specific versions of certain programs and packages. That way, you always have a clean environment and avoid issues with different programs wanting different versions of the same package.

Creating an environment is easy. We have already made one for you with Biopython, but you can see how easy it is below.

To create an environment named 'MyEnv' with Python version 3.6:

```
$ conda create -n MyEnv python=3.6                    Unix
```

To activate the environment:

```
$ conda activate MyEnv                                Unix
```

To deactivate the environment when finished:

```
$ conda deactivate MyEnv                              Unix
```

Running Python in the command line

Running a Python script through the command line is like running a bash script. You can add a shebang line in the first line of the script so the interpreter knows how to read it:

```
#!/usr/bin/env python3                                Python
```

Or you can run it using the Python command:

```
$ python myfirst.py                                   Unix
```

In this module, we will run the scripts using the IDE (Integrated development environment) called Spyder, from where any python script can be directly executed without needing a command line or shebangs.

Note: Python scripts have the extension '.py'.

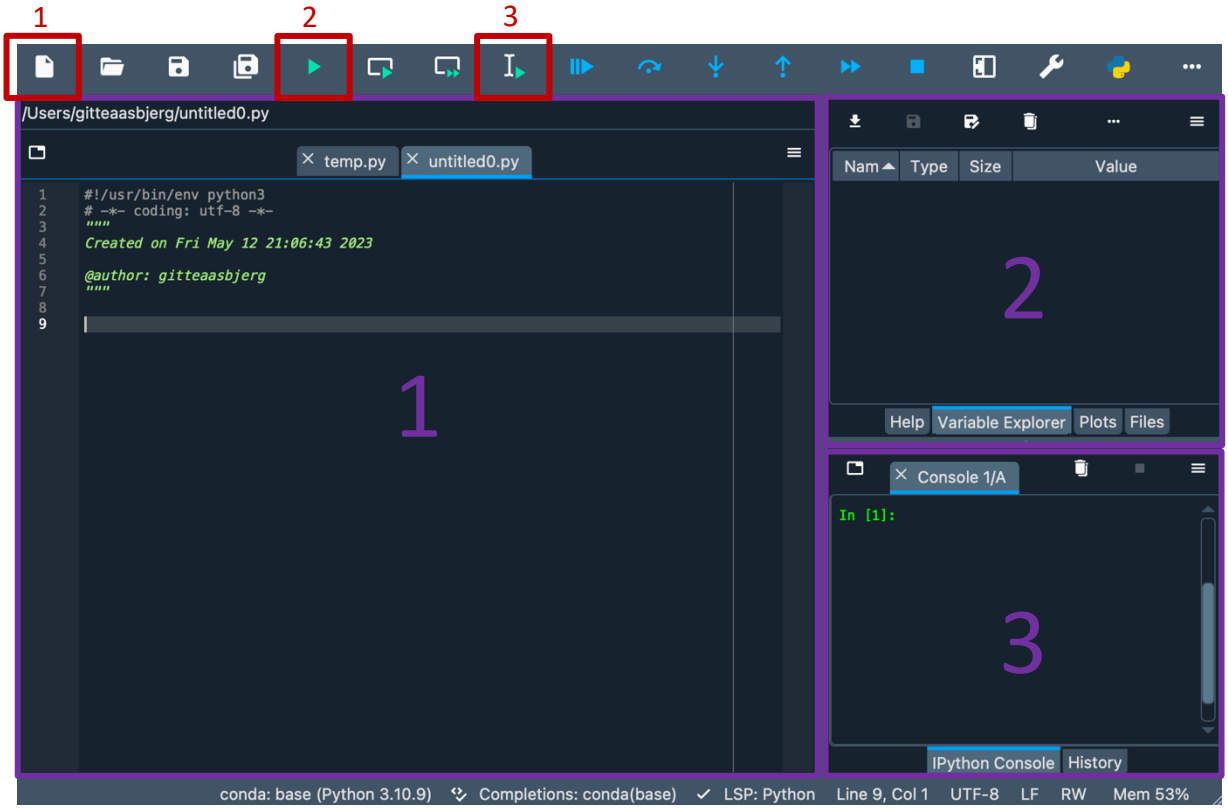
Spyder

The Spyder interface has three main panes:

- Pane 1: The Editor pane provides a robust code editing environment with features like syntax highlighting.
- Pane 2: The Variable Explorer pane provides a comprehensive overview of the assigned variables, allowing you to inspect their values, types, and dimensions.
- Pane 3: The IPython Console pane allows you to interactively run Python code, view output, and access a rich set of debugging and profiling tools.

The toolbar in the Spyder interface also has three handy buttons:

- Button 1: The New File button creates a new script or file
- Button 2: The Run button executes the current script, allowing you to run your code and observe the output.
- Button 3: The Run button executes the current selection, allowing you to run chunks of your code and observe the output.



Variables

Variables are containers used to store data and give them a name. Variables can hold various types of data, such as numbers, strings, or even more complex structures like lists. By using variables, you can perform computations, store user input, or represent information in a flexible and dynamic manner, enhancing the functionality and versatility of your Python programs.

You assign variables in Python using '=':

```python
VarA = "The variable VarA now contains a string"
VarB = 300
```

Note: when assigning strings to variables, you can use single or double quotes; it will ultimately have the same result. Variables are case-sensitive, so the variable **varA** differs from **VarA**. If you assign a new value to an existing variable, it will be overwritten.

You must define variables in the script before using them in a command. You can see all assigned variables in pane 2 (variable explorer). If you have not executed your script, the variables you have assigned in the scripts will not be displayed in the variable explorer.

Print

The print() statement in Python is a useful tool for displaying output to the console. It is a fundamental command that helps you observe and understand what is happening in your program at different stages of execution. It allows you to easily show text, variables, or expressions on the screen while your program is running. By using the print statement, you can debug your code, track the values of variables, and communicate information to the user.

Printing in Python is simple:

```python
print("This is a printout")
output: This is a printout

excuse = "My boyfriend ate my dog, so, I couldn't join the meeting."
print(excuse)
output: My boyfriend ate my dog, so, I couldn't join the meeting.

A = "Vodka"
B = "Cake"
print("Is",A,"and",B,"good for you?")
output: Is Vodka and Cake good for you?

print("Is"+A+"and"+B+"good for you?")
output: IsVodkaandCakegood for you?
```

Note: by default, separating the input of a print statement with a comma will add space between each input, whereas plus will not.

Commenting

If you want to add a comment in Python, it is like bash, where you use '#'.

```python
# This is a comment that will not run
print("This is not a comment")
# you can write anything you want here; it is not executable

output: This is not a comment
```

You can also comment out multiple lines of code. This is especially useful for code chunks you do not need to run now. To start and end a multi-line comment, use """ (triple quotes).

```python
"""
This code that is ignored
print(“This statement isn’t going to happen”)
"""
print("This will be printed, and is not commented out")

output: This will be printed, and is not commented out
```

Equal/not equal

The comparison operator “==” is used check whether one thing is equal to another. Not to get it confused with the “=” which is used to assign variables. Below is a table with other comparison operators:

Meaning	Operator
Equal	==
Not equal	!=
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=

Types

Python has several built-in data types that are fundamental for storing and manipulating different kinds of data. Some common data types in Python include:

1. Integers (**int**): Used to represent whole numbers without decimal points.
2. Floating-Point Numbers (**float**): Used to represent numbers with decimal points or fractional values and is accurate up to 15 decimals.
3. Strings (**str**): Used to represent sequences of characters, such as text or words, enclosed in single or double quotes.
4. Booleans (**bool**): Used to represent either True or False values, which are essential for logical operations and decision-making.
5. Lists: Used to store collections of items, which can be of different data types, and can be modified (mutable).
6. Tuples: Similar to lists, but they are immutable, meaning their elements cannot be changed once defined.
7. Dictionaries: Used to store key-value pairs, allowing you to retrieve values based on their associated keys.
8. Sets: Used to store unique elements in an unordered manner, providing operations like union, intersection, and difference.

the type() function is used to determine the type or class of an object. It returns the class or type of the object as a result. Here are some examples:

```python
type(3)          output: int
type(3.0)        output: float
type("True")     output: str
type(True)       output: bool
```

Python provides several functions to convert one type to another, such as int(), float(), str(), list(), tuple(), dict(), and bool(). Here are some examples:

```python
# Convert to str
str(3)      output: "3"

# Convert to float
float(3)    output: 3.0

# Convert to int
int(3.0)    output: 3
```

Boolean

A Boolean is a data type with two values: True and False. Booleans are used for logical operations and decision-making. You can create Boolean expressions using comparison operators like == and logical operators like *and*, *or*, and *not*. Here are some examples:

```python
# Boolean Expression and Output
(1 > 2)     # output: False
(1 == 2)    # output: False
(1 < 2)     # output: True

# Logical Operators:
(1 == 2 and 1 < 2) # output: False
(1 == 2 or 1 < 2)  # output: True
(not(1 > 2))       # output: True
```

Lists

A list is a versatile and commonly used data structure that allows you to store collections of items. It is created by enclosing comma-separated values or objects within square brackets '[ ]'. Lists can contain elements of different types, such as integers, strings, or even other lists. You assign a list as you assign variables, with an '='. Use proper names when assigning a list, and don't call a list for 'list'. Here are some examples:

```python
# Create list containing five strings
activities = ["hiking", "biking", "drinking", "murdering"]

# Create list containing multiple element types
mixed = [1, 1.0, ["list", "inside", "list"], True]
```

Python includes built-in methods to perform various operations on lists, such as append(), insert(), remove(), sort(), reverse(), len(), and more. Here are a few examples:

```
# Append element to list
activities.append("baking")
print(activities)
# output: ["hiking", "biking", "drinking", "murdering", "baking"]

# Remove element from list
activities.remove("murdering")
print(activities)
# output: ["hiking", "biking", "drinking", "baking"]

# Find length of list
len(activities)
# output: 4
```

Indexing

Elements within a data structure, such as lists, are accessed using **zero-based** indexing. Zero-based indexing means that the first element in a list is at index 0, the second element is at index 1, and so on. You can use square brackets [ ] and the index of the element to retrieve its value. Here are some examples:

```
# Create list                                                    python
activities = ["hiking", "biking", "drinking", "murdering"]

# access elements in list
activities[0]    # Output: hiking
activities[2]    # Output: drinking
```

Lists are mutable, meaning you can change the value of individual elements by assigning new values to specific indices:

```
# change element in list                                         python
activities[3] = "baking"
print(activities)
output: ["hiking", "biking", "drinking", "baking"]
```

Slicing

Slicing is a way to extract a portion of a list, string, or other sequence-like objects. It allows you to create a new sequence containing a subset of elements from the original sequence. In the table below, **start specifies the starting index** of a slice, and stop the ending element of a slice.

Slicing command	**Effect**
Object[start:stop]	Items from start to stop -1
Object[start:]	Everything from the start
Object[:stop]	Everything from beginning to stop -1

Object[:]	Everything
Object[-2:]	Last two items in the object
Object[:-2]	Everything besides the last two items

Here are some examples:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Extract a slice from index 2 to 5 (exclusive)
my_list[2:5]      # output: [3, 4, 5]

# Extract a slice from index 3 to the end (exclusive)
my_list[3:]       # output: [4, 5, 6, 7, 8, 9, 10]

# Extract a slice from the beginning to index 6 (exclusive)
my_list[:6]       # output: [1, 2, 3, 4, 5, 6]
```

If statements

If statements are used to add conditional execution of a statement. Such that, only if specific requirements are met will the code be executed. A simple *if statement* is shown below:

```
# if <expression> is TRUE: run <statement1> & <statement2>          python
if <expression>:
      <statement1>
      <statement2>
<statement3>
```

Only if <expression> is TRUE will <statement1> and <statement2> run; note that <statement3> will run regardless of <expression> being TRUE. That is because <statement1> and <statement2> have an indentation that indicates that it is part of the *if statement*. Note the ':' after <expression> to indicate the start of the *if statement*.

Unlike bash, there is no direct indication of the end of the *if statement*. The end of the *if statement* is indicated by the first new line without indentation compared with the beginning of the if statement. Here is an example:

```
# Start of if statement                                             python
if x > 69:
      print("I guess X is bigger than 69!")
      print("Damm…", x, "that's big!")
      # the indentation stops here and so does the if statement
print("Oh no, it's the end of the example")
```

Like bash, python also allows you to add an *else statement*, which only triggers if the 'if condition' is FALSE, or in other words, if 'if <expression>' is not TRUE.

You can also add more than two outcomes by adding an *elif statement* (else if). In the example below, you can see the general *if statement* with an additional *elif* and *else statement*. First, <expression1> is evaluated; if TRUE, <statement1> is executed; if not, then <expression2> is evaluated. If <expression2> is TRUE, then <statement2> is executed; if FALSE, then <statement3> in the else statement is executed instead. <statement4> is executed regardless of the outcome of the *if/elif/else* statements.

```python
if <expression1>:
      <statement1>
elif <expression2>:
      <statement2>
else:
      <statement3>
<statement4>
```

Making an accurate *if statements* can be tricky. Using the correct order of expressions along with the right comparison operators are a requirement to an accurate output. Here are some general examples and some pitfalls:

```python
# Initiate if statement
if X < 20:
      print(X, "is smaller than 20")
elif X < 10:
      print(X, "is smaller than 10")
else:
      print(X, "is bigger than 20")

   1) input: X = 15          output: 15 is smaller than 20
   2) input: X = 5           output: 5 is smaller than 20
   3) input: X = 20          output: 20 is bigger than 20
```

1) The first expression (if) is True as 15 is indeed smaller than 20. Lucky.
2) The first expression (if) is also True as 5 is indeed smaller than 20. However, the second expression (elif) would have been more fitting, but the elif expression is never reached as the first if expression is True. Here, changing the order of the first two expressions would make a more correct outcome.
3) Neither the first expression (if) nor the second (elif) is True – thus the last expression (else) is run. However, this statement is not correct either. Using different comparison operators would make a more accurate outcome.

For loops

For loops in Python are used to iterate over a sequence of elements, such as a list, string, or range of numbers, executing a block of code for each element in the sequence. The loop iterates over each item one by one, allowing you to perform repetitive tasks or operations on them. A simple *for loop* is shown below:

```python
for <var> in <iterable>:
      <statement(s)>
```

The <statement(s)> will be executed for each item in <iterable>. Here <iterable> is the collection of objects, such as a list. <var> is a loop variable; it takes the value of the current item in <iterable> and will take the value of the next item in <iterable> each time it goes through the loop. The loop ends when all elements in <iterable> have gone through the statement. The <statement(s)> are in the loop body because of the indentation as with *if statements*; the indentation is essential to note what is part of the loop.
Note that you can call <var> whatever you like. It will always have the value of an item in the loop. Here is an example:

```python
# Create list
list_activities = ["hiking ", "murdering ", "drinking"]

# Iterate through every item in list_activities
for activity in list_activities:
      print(activity,"is a wholesome activity!")

output:     hiking is wholesome activity
            murdering is a wholesome activity
            drinking is a wholesome activity
```

The break statement is used to exit or terminate a loop prematurely. It is primarily used within *for loops* and *while loops* to interrupt the loop's execution before it reaches its normal end. Here is an example:

```python
# Iterate through every item in list_activities
for activity in list_activities:
      if activity == "murdering":
            print("Mate… No")
            print("I… Gotta go")
            break
      else:
            print("Let's go", activity)

output:     Let's go hiking
            Mate… No
            I… Gotta go
```

Note: the loop was terminated before iterating over "drinking".

Counting

You can use a counting variable to count how many times you go through a loop. Set "counter = 0" before a loop and add "counter += 1" inside the loop; this will add 1 to "counter" every time you go through the loop. You can also use -= 1. This is the opposite of += and subtracts 1 for every iteration through the loop. Here is an example:

```python
# initiate counter
counter = 0

# Iterate through every item in list_activities
for stuff in list_activities:
      print(“Should we go”, stuff, “today?”)
      counter += 1 # add 1 for every iteration
print(“We have”, counter, “options of activities today”)

output:     should we go hiking today?
            Should we go murdering today?
            Should we go drinking today?
            We have 3 options of activities today
```

File handling in Python

File handling in Python involves working with files to read or write data. Python provides a set of functions and methods that make it easy to handle files. To open a file, you can use the open() function. It takes the file path and the mode as parameters. The mode specifies whether you want to read, write, or append to the file:

```python
file = open("example.txt", "r")      # read mode
file = open("example.txt", "w")      # write mode
file = open("example.txt", "a")      # append mode
```

To write data to a file, open the file in write mode ("w") or append mode ("a"), and then use the write() method to add content:

```
file = open("example.txt", "w")    # creates a new file or overwrites the existing
file.write("Overwriting the file!") # overwrites the file with the given content
file.close()                       # closes the file after writing
```

Biopython

Biopython is the largest and most popular bioinformatics package for Python. It contains several different sub-modules for everyday bioinformatics tasks. Biopython provides an easy way to parse FASTA files and extract sequence data. You can use the SeqIO.parse() function to read the FASTA file and iterate over its sequences. Here is an example:

```python
# Loading required packages
from Bio import SeqIO
from Bio.Seq import Seq

# Identifying fasta file
filename = "path/to/file/sequences.fasta"

# Parsing the FASTA file
sequences = SeqIO.parse(filename, "fasta")

# Iterating over the sequences
for sequence in sequences:
    print("ID:", sequence.id)            # print sequence ID
    print("Sequence:", sequence.seq)     # print sequence
    print("Length:", len(sequence))      # print sequence length
```

System arguments

System arguments allow you to pass input values or parameters to a script or program directly from the command line. The arguments are passed as strings and can be accessed within your Python script. To work with system arguments, you need to import the sys module, which provides access to system-specific parameters and functions. Consider this example – let's call the script example.py:

```python
Import sys

# Accessing command-line arguments
arg1 = sys.argv[1]
arg2 = sys.argv[2]
arg3 = sys.argv[3]

# Using the arguments
print("Argument 1:", arg1)
print("Argument 2:", arg2)
print("Argument 3:", arg3)
```

Executing the script in the command line will give the following output:

```Unix
$ python example.py duck duck goose

Argument 1: duck
Argument 2: duck
Argument 3: goose
```

While loop

A *for loop* is a defined iteration, but a *while loop* might be preferred if you do not know how many iterations you need. A *while loop* is an indefinite iteration that stops when a given condition stops being TRUE.

```
while <expression>:                                                    python
      <statement(s)>
```

As with *for loops* and *if statements*, indentation is critical, and you can see that the <statement(s)> is part of the loop because it has an indentation. The <statement(s)> will run until the condition in <expression> is False. The <expression> usually involves one or more variables defined before the loop, which is then modified as you go through the loop. The *while loop* stops when the variables change, so the condition becomes False.

```
# Define number of cakes                                               python
cakes = 2

# Run if cake variable is bigger than zero
while cakes > 0
      cake -= 1 # subtract 1 for every iteration
      print("Kasper ate a cake")
print("Kasper ate all the cakes")

output:     Kasper ate a cake
            Kasper ate a cake
            Kasper ate all the cakes
```

For every iteration of the loop, Kasper will have eaten one cake. This *while loop* stops when "cake" has the value of 0. The variable in the <expression> must be a variable that changes through the loop and has the potential to make the condition False; if not, the while loop will continue at infinite.

The built-in function len() which finds the length of an object, such as number of characters in a string or elements in a list. This versatile function has many uses, including when you

```
# initiate counter                                                     python
i = 0

# Create list
list_activities = ["hiking ", "murdering ", "drinking "]

# Run if counter is smaller than list_activities
while i < len(list_activities):
      print(list_activities[i])
      i += 1 # add 1 for every iteration

output:     hiking
            murdering
            drinking
```

need a maximum, either when finding a range for slicing or iterating through a list using a *while loop*.

Functions

Functions are a fundamental concept that allows you to organize and reuse code efficiently. A function is a block of code that performs a specific task and can be called multiple times throughout your program. It takes input arguments (optional) and can return a value (optional) as a result.

```python
# Creating a function called greet that takes a name parameter
def greet(name):
    print("Hello, " + name + "!")

# Executing function
greet("Kasper")
output: Hello, Kasper!
```

Functions can also return values using the return statement. Here's an example:

```
# Creating a function that return values using the return statement
def add_numbers(a, b):
    return a + b

# Capture the returned value by assign a variable
VarA = add_numbers(1,2)
Print(VarA)
output: 3
```

Dictionaries

A dictionary is a powerful data structure that allows you to store and retrieve data using key-value pairs. It is also known as an associative array or a hash map in other programming languages. A single key can be associated by multiple values if values are assigned as lists. Here is an example:

```python
# Creating a dictionary of lists
activities = {"yes": ["hike"], "no": ["murder"]}

# Adding a new value to the list of values
activities["yes"].append("sing")
print(activities["yes"])
Output: ['hike', 'sing']

# Adding a new key and value (value is not a list, so appending is not possible)
activities["maybe"] = "sleep"
print(activities)
Output: {'yes': ['hike','sing'], 'no': ['murder'], 'maybe' : 'sleep'}

# Accessing a specific value from the list of values
first_activity = activities["yes"][0]
print(first_activity)
Output: 'hiking'
```