Python Handout

Table of Contents

Introduction

Welcome to the Python handout! This guide covers the essentials of Python scripting. Known for its simplicity and readability, Python is a powerful language used in web development, data analysis, and automation. With its rich libraries and strong community support, it's great for both beginners and experts. Let's dive into key topics to help you master Python!

Environments

Environments help keep projects organized by using specific versions of programs and packages, ensuring consistent behavior across different machines. **Conda** is a package and environment management system that allows you to install and switch between different library versions to avoid conflicts.

Setting up a Conda environment is easy! We've already created one for you, so you don't have to worry about it. But if you're curious, you can see how it is done below —just so you can see how simple it really is.

To activate the environment:

```
$ conda activate MyEnv                                    Unix
```

To deactivate the environment when finished:

```
$ conda deactivate MyEnv                                  Unix
```

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

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

Running Python from 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

<code>
```

And run it through the terminal like so:

```
$ ./myfirst.py                                            Unix
```

Or you can run it using the Python command:

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

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

In this module, we'll use Spyder, an Integrated Development Environment (IDE), to run Python scripts directly without a command line or shebangs.

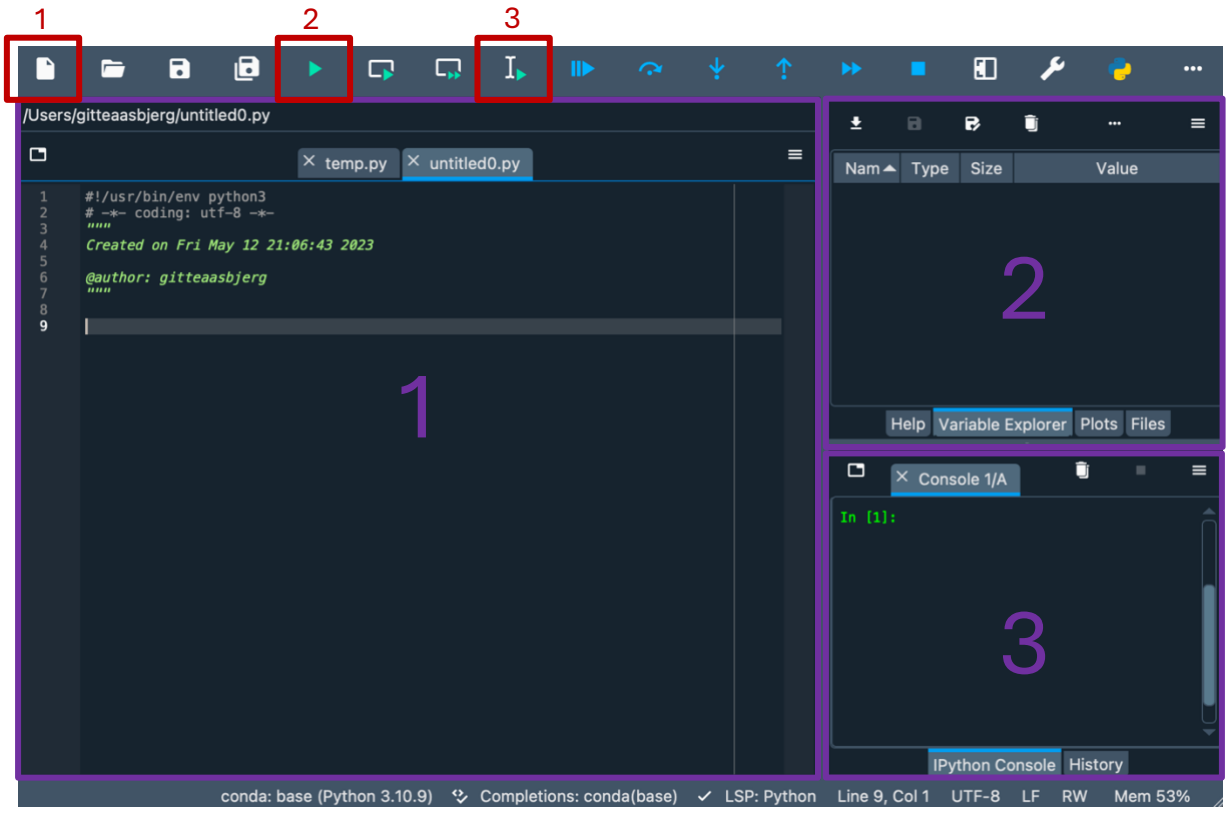
Spyder

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 displays assigned variables, showing their values, types, and dimensions.
- **Pane 3**: The **IPython Console** is used for running Python code interactively, viewing output, and accessing debugging tools.

The toolbar in the Spyder interface also has three handy buttons we would like to highlight:

- **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**: This Run button **executes the current selection**, allowing you to run chunks of your code and observe the output.



Simple concepts

Variables

Variables store data and can hold different types, such as numbers, strings, or lists. They allow you to perform calculations, store user input, and manage information dynamically, making your Python programs more flexible and functional.

In Python, you assign variables using =, just like in Bash:

```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.
Note: 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.

Variables must be defined in the script before being used in a command. If the script hasn't been executed, assigned variables won't appear in the Variable Explorer (Pane 2), where all defined variables are displayed.

Print

The print() statement in Python displays output in the console. You can use it to display text, variables, or expressions, making it a valuable tool for understanding your program's execution.

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 + will not.

Commenting

To add a comment in Python, use #, just like in Bash:

```python
# This is a comment that will not run
This is not a comment and will cause syntax error
# you can write anything you want here; it is not executable

output: SyntaxError: invalid syntax
```

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 is ignored
print("This statement isn't going to happen")
"""
print("This will be printed, and is not commented out")
output: This is not commented out and will be printed, and
```

Data 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**): Whole numbers without decimal points.
2. **Floating-Point Numbers** (**float**): Numbers with decimal points.
3. **Strings** (**str**): Sequences of characters, such as text or words, enclosed in quotes.
4. **Booleans** (**bool**): True or False values.
5. **Lists**: Ordered, modifiable collections of items of any data type.
6. **Sets**: Unordered collections of unique elements.
7. **Dictionaries**: Key-value pairs for fast data lookup and retrieval.

The type() function is used to determine the type of an object. Here are some examples:

```python
type(3)                    output: int
type(3.0)                  output: float
type("True")               output: str
type(True)                 output: bool
type([1, "hello", 3.5]) output: list
type({1, "world", 3.5}) output: set
type({"a": 1, "b": 2})  output: dict
```

Python provides several functions to convert one data type to another, such as int(), float(), str(), list(), 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. Comparison operators check the relationship between values and return a Boolean result. Here are the comparison operators in Python:

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

Logical operators are used to perform logical operations on Boolean values. These operators allow you to combine Boolean values and expressions to evaluate complex conditions. The three main comparison operators include:

- **and**: Returns True if both operands are True.
- **or**: Returns True if at least one of the operands is True.
- **not**: Negates a Boolean value, meaning it flips True to False and False to True.

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 flexible way to store multiple items. You create one using square brackets [ ], separating items with commas. Lists can hold different types of data, like numbers, strings, or even other lists. Assign a list using = just like variables, but avoid naming it "list.". Here are some examples:

```python
# Create list containing four 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 .len(), .append(), .insert(), .remove(), .sort(), .reverse(), and more. Here are a few examples:

```python
# 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

In Python, elements in a list are accessed using **zero-based indexing**, meaning the first item is at index 0, the second at 1, and so on. Use square brackets [ ] with the index number to get a specific value. Here are some examples:

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

# 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:

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

Slicing

Slicing extracts a portion of a list, string, or sequence, creating a new subset. In the table below, the **start** index defines where the slice begins, and the **stop** index defines where it ends (excluding the stop index itself):

Slicing command	Effect
Object[start:stop]	Items from start to stop
Object[start:]	Everything from the start
Object[:stop]	Everything from beginning to stop
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
my_list[2:5]        # output: [3, 4, 5]

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

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

Remember that python uses zero-based indexing.

If statements

If statements allow conditional execution, meaning code runs only if a specific condition is met. A basic if statement looks like this:

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

- <statement1> and <statement2> run **only if** <expression> is **True**.
- <statement3> runs regardless, as it is not indented under the if block.
- The colon (:) after <expression> marks the start of the if statement.
- Unlike Bash, Python does not use an explicit end marker; the if-block ends when indentation returns to the previous level.

Here is an example:

```python
# Start of if statement
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")

input: X = 80        output:  I guess 80 is bigger than 69!
                              Damn… 80 that's big!
                              Oh no, it's the end of the example

input: X = 20        output:  Oh no, it's the end of the example
```

Note: the last print statement is printed regardless of the value of x.

Elif and Else statements

You can add more than two outcomes using elif (else if) and else statements:

- **if** runs when its condition is True.
- **elif** checks another condition if the if condition was False.
- **else** runs only if all previous conditions were False.

Here is a generel example:

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

- <expression1> is evaluated first. If True, <statement1> runs.
- If False, <expression2> is checked. If True, <statement2> runs.
- If both are False, <statement3> in the else-block runs.
- <statement4> runs regardless of the conditions above.

Key points:

- Order matters—conditions should be structured logically for accurate results.
- Use appropriate comparison operators to avoid unexpected behavior.

Below are examples and common pitfalls to watch out for:

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

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

1. **CORRECT**: X = 15 works correctly because 15 < 20 is True, so the i- block runs.
2. **INCORRECT**: Y = 5 satisfies Y < 20, so the elif Y < 10 condition is never checked.
   Fix: Swap the order of elif X < 10 and if X < 20 to ensure smaller numbers are categorized correctly.
3. **INCORRECT**: X = 20 triggers else, but the statement "20 is bigger than 20" is incorrect.
   Fix: Change the comparison to X <= 20 or adjust the final message.

For loops

For loops 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 loop goes through each <var> in <iterable> one by one. The indented statements inside the loop run for every item. 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 a wholesome activity
                murdering is a wholesome activity
                drinking is a wholesome activity
```

Note: that you can call <var> whatever you like. It will always have the value of an item in the loop.

The break statement is used to exit or terminate a loop prematurely. 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 last item.

Counting

You can use a counting variable inside a for-loop to track how many times the loop runs.

- Set counter = 0 before the loop.
- Use counter += 1 inside the loop to increase the count each time.
- You can also use counter -= 1 to decrease the count instead

```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. To open a file, you can use the open() function that takes two parameters: the file path and the mode. 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 most widely used bioinformatics package for Python, offering multiple sub-modules for common bioinformatics tasks. It provides a simple way to parse FASTA files and extract sequence data using SeqIO.parse(). Here is an example:

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

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

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

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

Why use Biopython:

- Easily reads FASTA, GenBank, and other bioinformatics formats
- Provides built-in tools for sequence analysis, alignment, and annotations
- Simplifies computational biology workflows for researchers and scientists

System arguments

System arguments allow you to pass input values or parameters directly to a script from the **command line**. These arguments are passed as **strings** and can be accessed within your Python script. To use system arguments, **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
```

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

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

```python
# Using the arguments
print("Argument 1:", a)
print("Argument 2:", b)
print("Argument 3:", c)
```

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

While loop

A for loop runs a fixed number of times, but a while loop is used when you don't know how many iterations are needed. A while loop continues running as long as its condition remains True. Here is a general example:

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

- Indentation is essential, just like in for-loops and if-statements.
- The loop runs until <expression> becomes False.
- The condition usually involves a variable that changes within the loop to eventually stop it; if not, **the while loop will continue at infinite**.

```python
# Define number of cakes
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
```

- *cake* starts at 2.
- The loop runs as long as *cake* > 0.
- *cake* decreases by 1 each time.
- Once *cake* reaches 0, the condition becomes False, and the loop stops.

The built-in function len() returns the length of an object, such as the number of characters in a string or elements in a list. This versatile function has many uses, including when you need a maximum, either when finding a range for slicing or iterating through a list using a while-loop. Here is an example:

```python
# initiate counter
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
```

Functions

Functions help organize and reuse code efficiently by performing specific tasks that can be called multiple times. A function can take input arguments (optional) and return a value (optional). Defining a function in Python:

- Use the **def** keyword.
- Provide a function name and parameters inside parentheses.
- The function body is indented.
- Call the function to execute it.

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

# Executing function
greet("Kasper")   # output: Hello, Kasper!
greet("neighbor") # output: Hello, neighbor!
greet("7")        # output: Hello, 7!
```

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

```python
# 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
```

- The returned value is stored in VarA and can be used later.
- The function is reusable for any two numbers.

Sets

A set is a mutable, unordered collection of unique elements in Python. It is defined using curly braces {} and can hold different data types, such as numbers and strings. Key features of sets:

- **No duplicates**: If an element is added multiple times, it appears only once.
- **Unordered**: Elements have no fixed position, so indexing is not possible.
- **Mutable**: You can add or remove elements after creation.

```
# Creating a set
my_set = {1, 2, 7, 3, 4, 2, 5, 4, 2, 1, 1}
print(my_set)                # output {1, 2, 3, 4, 5, 7}

# Adding elements to a set
my_set.add(6)
print(my_set)                # output {1, 2, 3, 4, 5, 6, 7}

# Removing elements from a set
my_set.remove(3)
print(my_set)                # output {1, 2, 4, 5, 6, 7}
```

Dictionaries

A dictionary in Python is a powerful and flexible data structure that stores data as key-value pairs {key: value}. Unlike lists, where elements are accessed by index, dictionaries allow fast lookups using unique keys, making data retrieval more efficient.

```
# Creating a dictionary of lists                                    python
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 not a list, appending 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: 'hike'
```

Dictionaries are useful for:

- Perfect **for structured data** like user profiles, config-urations, and settings.
- Retrieving a value using a key is **much faster** than searching through a list.
- Can hold **multiple data types**, including lists, tuples, sets, or even other dictionaries (nested dictionaries).
- You can easily add or modify key-value pairs at runtime.