Handout: Advanced Python

Table of Contents

Escape characters

Escape characters in Python are special characters preceded by a backslash (\) that are used to represent characters that are difficult to type directly. Using escape characters allows you to include special characters in strings and control the formatting of text within your Python code. Here are some commonly used escape characters in Python:

Escape character	Represents
\n	Newline
\t	Tab
\\	Backslash
\’	Single quote
\”	Double quote

Here is a simple example with multiple escape characters:

```
# print string with multiple escape characters                                   python
print('Hi!\nAllright, you don\'t have to ignore me…')
output   Hi!
               Alright, you don't have to ignore me…'
```

String manipulations

String manipulation refers to modifying, formatting, searching, and extracting data from strings. Python offers a variety of built-in methods to perform these tasks efficiently. Some common string manipulation techniques include:

- **Split**: Splits a string into a list of substrings based on a delimiter.
- **Strip**: Removes leading and trailing whitespace characters from a string.
- **Join**: Joins elements of a list into a single string using a specified separator.
- **Upper**: Converts all characters in a string to uppercase.
- **Lower**: Converts all characters in a string to lowercase.
- **Replace**: Replaces occurrences of a specified substring with another substring.
- **Find**: Returns the index of the first occurrence of a specified substring within the string.
- **Startswith**: Checks if the string starts with a specified substring.
- **Endswith**: Checks if the string ends with a specified substring.

The sections below will focus on the manipulations needed for today’s exercises.

Split

split() breaks a string into a list of substrings based on a specified separator. By default, the separator is whitespace characters (spaces, tabs, newline characters), but you can specify any character or string as the separator. Here is a simple example:

```
# split string into list at every whitespace                    python
message = 'I am so sorry. I ate your homework'
print(message.split())
output   ['I', 'am' 'so', 'sorry.', 'I', 'ate', 'your', 'homework']
```

You can also specify a custom separator within the split() method:

```
# split string into list at every 'o'                           python
message = 'I am so sorry. I ate your homework'
print(message.split('o'))
output   ['I am s', ' s', 'rry. I ate y', 'ur h', 'mew', 'rk']
```

Strip

strip() removes leading and trailing whitespace characters from a string. It does not modify the original string but returns a new string with the whitespace characters removed:

```
# strip string from leading and trailing whitespace             python
rudecomment = '      Sorry I wasn\'t listening...   '
print(rudecomment.strip())
output   'Sorry I wasn't listening...'
```

Removing whitespace is the default behavior, you can also remove other characters, however, only characters that are leading or trailing. Here is an example:

```
# strip string from leading and trailing whitespace             python
influencerspeak = '#blessed #sunshine #morningcoffee'
print(influencerspeak.strip())
output   'blessed #sunshine #morningcoffee'
```

Tuples

A tuple is an immutable sequence, meaning its elements cannot be changed after creation. Tuples are ordered collections that can hold different data types and are enclosed in parentheses (). Key Features of Tuples:

- **Immutable** – Elements cannot be added, removed, or modified.
- **Ordered** – Maintains the order of elements.
- **Allows duplicates** – Unlike sets, tuples can store repeated values.
- **Indexable** – Access elements using indexing, just like lists.
- **Efficient** – Faster than lists for fixed data storage.

While you cannot modify items in a tuple, you can modify internal lists. Here is an example:

```
# Create a tuple containing three strings                              python
healthy_dinner = ("pizza", ["chicken burger", "beef burger"], "Shawarma")

# Modify internal list in tuple
healthy_dinner[1][1] = "cheese burger"
print(healthy_dinner):
output('pizza', ['chicken burger', 'cheese burger'], 'Shawarma')
```

List manipulations

List manipulation commands in Python enable the modification, manipulation, and access of list elements. These commands offer flexibility in managing and manipulating list data structures. Some common list manipulation commands include:

- **extend**: Appends elements from another list to the end of the current list.
- **append**: Adds an element to the end of the list.
- **insert**: Inserts an element at a specified position in the list.
- **remove**: Removes the first occurrence of a specified element from the list.
- **pop**: Removes and returns the element at a specified index in the list.
- **index**: Returns the index of the first occurrence of a specified element in the list.
- **count**: Returns the number of occurrences of a specified element in the list.
- **sort**: Sorts the elements of the list in ascending or descending order.
- **reverse**: Reverses the order of elements in the list.
- **slice**: Extracts a portion of the list based on specified indices.
- **len**: Returns the number of elements in the list.

Extend

Combining lists involves **merging the elements of one list with another** to create a single, larger list. There are several ways to combine lists in Python, one of which is the **extend()** method:

```
# Create two lists of strings                                                   python
cakes = ['cheesecake', 'fishcake']
fruits = ['cherry', "plum", 'eggplant']

# Combine the lists by using the + operator
desserts = cakes + fruits
print(desserts)
output ['cheesecake', 'fishcake', 'cherry', 'plum', 'eggplant']

# Combine the lists by using the extend command
desserts = cakes.extend(fruits)
print(desserts)
output ['cheesecake', 'fishcake', 'cherry', 'plum', 'eggplant']
```

Append

append() is used to **add an element to the end of a list**. This method is particularly useful when you want to dynamically add elements to a list without having to specify the index where the element should be inserted. Here is a simple example:

```python
# Append string to end of desserts list
desserts.append('tomato')
print(desserts)
output ['cheesecake', 'fishcake', 'cherry', 'plum', 'eggplant', 'tomato']
```

Count

count() is used to **count the number of occurrences** of a specified element in a list. When called on a list, count() takes a single argument, which is the element to be counted. It returns the number of times that element appears in the list. Here is a simple example:

```python
# Make list called stuffIAteToday
stuffIAteToday = ['chocolate', 'bananas', 'chocolate', 'tomato soup']

# Count and print the amount of times chocolate is in list
print(stuffIAteToday.count('chocolate'))
output 2
```

Replace

Lists don't have a built-in **replace()** method, however, you can achieve a similar result by using indexing to replace elements within a list. Here is a simple example:

```python
# Replace string at index 1 with cookie
desserts[1] = 'cookie'
print(desserts)
output ['cheesecake', 'cookie', 'cherry', 'plum', 'eggplant', 'tomato']
```

Remove

remove() **deletes the first occurrence of a specified value** from a list. It takes a single argument, which is the value to be removed from the list.

```python
# Make list called stuffIAteToday
stuffIAteToday = ['chocolate', 'bananas', 'chocolate', 'tomato soup']

# Remove first occurance of chocolate
print(stuffIAteToday.remove('chocolate'))
output   ['bananas', 'chocolate', 'tomato soup']
```

Alternatively, the **pop()** method removes and returns the element at a specified index from a list. This method takes an optional argument, which is the index of the element to be removed:

```python
# Make list called stuffIAteToday
stuffIAteToday = ['chocolate', 'bananas', 'chocolate', 'tomato soup']

# Remove first occurance of chocolate
stuffIAteToday.pop(2)
output   chocolate
```

Sort

sort() is used to **sort the elements of a list** in **ascending** order by default. It modifies the original list in place and does not return a new list. This method can also take optional arguments such as reverse=True to sort the list in **descending** order:

```python
# Make two lists comprised of numbers and strings
stuffIAteToday = ['chocolate', 'bananas', 'chocolate', 'tomato soup']
listOfNumbers = [4,7,2,6,1]

# Sort stuffIAteToday alphabetically
stuffIAteToday.sort()
print(stuffIAteToday)
output ['bananas', 'chocolate', 'chocolate', 'tomato soup']

# Sort listOfNumbers ascending
listOfNumbers.sort()
print(listOfNumbers)
output [1,2,4,6,7]

# Sort listOfNumbers descending
listOfNumbers.sort(reverse = True)
print(listOfNumbers)
output [7,6,4,2,1]
```

User input

User input refers to the mechanism through which a Python program can accept data or information from the user during runtime. The **input**() function is used to prompt the user to enter data, which can then be stored in variables or processed by the program. Here is a simple example:

```
# prompt user to enter name and print in sentence                python
name = input("Enter your name: ")
print("Hello, " + name + "!")

input:    Leo
output    Hello, Leo!
```

In this example, the input() function prompts the user to enter their name. The text "Enter your name: " serves as the prompt. Once the user enters their name and presses Enter, the input is stored in the variable name. The program then prints a greeting message using the entered name.

It's important to note that the input() function always returns a string, even if the user enters a number or another type of data. If you need to convert the input to a different data type, such as an integer or a float, you can use type conversion functions like int() or float(). Here is a simple example:

```
# prompt user to enter name and print in sentence                python
age = input("Enter your age: ")
age = int(age)  # Convert the input to an integer
next_age = age + 1
print("Next year, you will be", next_age, "years old."

Input:    44
Output    Next year, you will be 45 years old.
```

User input is a powerful feature that allows Python programs to interact with users dynamically, making programs more interactive and versatile. However, it's important to handle user input carefully to avoid errors or security vulnerabilities, such as input validation and sanitization.

Packages

A **module** is a collection of pre-written code that you can import and use instead of writing everything from scratch. **Packages** are libraries of multiple modules designed for specific tasks, making programming more efficient.

To use a module, you first need to have the package installed in your environment. Once installed, you can import the module and use its functions. Since we have already installed the Biopython package, you just need to activate the correct Conda environment to access it:

```
conda activate BTG_biopython
```

To use the code from a module, you need to **import it at the top of your script** so that the rest of your script can access it. You can import an entire package or specific modules within a package using the following syntax:

```
import module                #import entire modules                python
from module import object    #import specific objects from modules
import package as alias      #assign aliases to imported packages
```

If you are running your script in Spyder, you may need to update the Python path to ensure Spyder can locate installed packages. Go to Preferences → Python Interpreter → Select "Use the following Python interpreter" and browse to the correct environment (e.g., from Conda). Restart Spyder for the changes to take effect.

Biopython

Biopython is a comprehensive Python library built for computational biology and bioinformatics. It provides powerful tools for analyzing and processing biological data, including:

- DNA, RNA, and protein sequences
- Sequence alignments
- Structural biology (protein structures, molecular modeling)
- Parsing bioinformatics file formats (FASTA, GenBank, etc.)
- Phylogenetics and evolutionary analysis

Seq module

The Seq module in Biopython provides functionality for working with biological sequences, such as DNA, RNA, and proteins. It is a part of Biopython's Bio package and some key features of the Seq module include:

- **Creation of Sequence Objects**: The Seq class allows you to create sequence objects by providing a string representing the sequence of interest. These sequences can include nucleotide (DNA and RNA) or amino acid sequences.

- **Sequence Manipulation**: The module provides methods for manipulating sequences, such as reverse complementation, translation, transcription, back translation, and finding open reading frames (ORFs).

```python
# import module
from Bio.Seq import Seq

# define sequence
GFP_truncate = Seq("ATGAGTAAAGGAGAA")

print("Sequence: ", GFP_truncate)
print("Complement: ", GFP_truncate.complement())
print("Reverse complement: ", GFP_truncate.reverse_complement())
print("Transcribed to RNA: ", GFP_truncate.transcribe())
print("Translated to amico acids: ", GFP_truncate.translate())

output:  Sequence:  ATGAGTAAAGGAGAA
                    Complement:  TACTCATTTCCTCTT
                    Reverse complement:  TTCTCCTTTACTCAT
                    Transcribed to RNA:  AUGAGUAAAGGAGAA
                    Translated into amico acids:  MSKGE
```

SeqIO module

The SeqIO module in Biopython provides tools for reading and writing biological sequence files in various formats. It is part of Biopython's Bio package and simplifies the parsing and formatting of sequence data commonly used in bioinformatics. Key features of SeqIO:

- **File Format Support** – Reads and writes sequence data in multiple formats, including fasta, genbank, fastq, swiss-prot, and more.
- **Parsing Sequences** – Converts sequence data into Biopython's SeqRecord objects, which store biological sequences along with metadata (e.g., ID, description).

In the demonstration below, we will use the FASTA file "overpriced_garden.fasta" as an example:

```
>Seq1_Orchid [organism=Phalaenopsis equestris var. leucaspis]
CCTATACCTAATTTTCGGCGCATGAGCCGGAATGGTGGGTACCGCTCTAAGCCTCCTCATTCGAGCAGAA
CTAGGCCAACCCGGAGCCCTTCTGGGAGACGACCAAGTCTACAACGTGGTTGTCACGGCCCATGCCTTCG

>Seq2_Petunia [organism=Petunia integrifolia subsp. inflata]
TAGTTGGAACAGCCCTCAGCCTACTCATCCGAGCAGAACTAGGCCAACCCGGAACCCTCCTGGGAGATGA
CCAAATCTACAATGTAATCGTCACTGCCCATGCCTTCGTAATAATCTTCTTCATAGTAATACCAGTCATA
```

```
# import module                                                                                    python
from Bio import SeqIO

# open and print ID's from fasta
for record in SeqIO.parse("path/to/overpriced_garden.fasta", "fasta"):
    print(record.id)

output:  Seq1_Orchid
                Seq2_Petunia
```

Note: One of the more useful functions of Biopython is **id().** Use it to get the ID of the sequence.
Note: When using the built-in **open()** function, the **'with'** statement ensures the file is

```
# import module                                                                                    python
from Bio import SeqIO

# open and print ID's from fasta
with open("path/to/overpriced_garden.fasta") as handle:
    for record in SeqIO.parse(handle, "fasta"):
        print(record.id))

output:  Seq1_Orchid
                Seq2_Petunia
```

automatically closed after reading

gzip module

The gzip module enables compression and decompression of gzip-compressed files, commonly used to reduce file size. It requires two inputs: the file name and the mode. In this course, we use "rt" mode only ("r" for **reading** and "t" for **text** mode). Here is an example:

```
# import module                                                                                    python
from Bio import SeqIO
import gzip

# define path to compressed fasta file
plants = "path/to/overpriced_garden.fasta.qz"

# open and print ID's from fasta
with gzip.open(plants, "rt") as handle:
    for record in SeqIO.parse(handle, "fasta"):
        print(record.id)

output:  Seq1_Orchid
                Seq2_Petunia
```

sys module

The sys module provides access to system-specific parameters and functions and is part of the Python Standard Library. It is commonly used for system interaction, command-line arguments, and environment variables.

In this course, we will use **sys.argv** to pass command-line arguments to Python scripts. Like gzip, you don’t need to specify its source—simply import the module to use it.

Running the following script in the terminal:

```
# import module                                                              python
import sys

pain = str(sys.argv[1])
print("Hello cruel world!")
print("Today’s pain measure is " + pain + ".")
```

Would have the following output:

```
./hello_cruel_world.py 9000                                                  unix
output:   Hello cruel world!
          Today’s pain measure is 9000.
```