BINF-5007

Using external resources and common Python errors

Syntax Errors

- Occur when the Python interpreter encounters incorrect or unexpected code structure
- Common causes include:
  - Misspelled keywords (e.g., pritn instead of print)
  - Incorrect indentation
  - Missing colons after if, for, while and similar statements
  - Unmatched parenthesis

SYNTAX ERRORS

```
1    pritn("Hello World") #Should be print
2
3    if True:
4    print("Something") #Should be indented
5
6    if True #Missing colon
7        print("Something else")
8
9    round(input("Enter a number:") #Missing one parenthesis
```

How to fix syntax errors

- Read error Messages
  - Python provides detailed error messages indicating the type and location of the error
- Check code structure
  - Ensure proper indentation, correct spelling, and matching parentheses
- integrated development environments (IDE) like PyCharm or VSCode can help identify and correct syntax errors

SYNTAX ERRORS

WHAT ERRORS CAN YOU FIND IN THIS CODE?

```
1    def calculate_velocity(displacement: float, time: float)->float
2    vel:displacement/time
3    return vel
```

Runtime Errors

- Occur while the program is running, causing it to crash or behave unexpectedly.
- Some possible causes:
  - Division by zero
  - Invalid operations like adding a string to an integer
  - accessing undefined variables
  - File handling errors (e.g., trying to open non-existent file)

RUNTIME ERRORS

```
1     num = 10
2     den = 0
3     result = num/den #Division by zero
4
5     result = "Hello"+num #Invalid operation (TypeError)
6
7     print(other_var) #NameError because 'other_var' is not defined
8
9     with open('./path/to/no/existing/file.txt', 'r') as fh:
10        contents = fh.readlines() #FileNotFoundError
```

How to fix runtime errors

- Validate inputs: ensure that inputs are valid and appropriate for the operations being performed.
- Test thoroughly: regularly test your code to catch and fix runtime errors early.
- Use exception handling: implement try and except blocks to handle potential errors

```
num = 10
den = 0
try:
    result = num/den #Division by zero
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

RUNTIME ERRORS

WHAT IS WRONG WITH THIS CODE?

```
1  current_year = input("Enter the current year: ")
2  toronto_age = current_year - 1793
3  message = "Toronto was founded in 1793. The city is "+toronto_age+" years old."
4  print(message)
```

Logical/Semantic Errors

- Occur when the code runs without crashing but produces incorrect or unintended results
- May be caused by:
  - Incorrect algorithm or Faulty logic
  - Misuse of variables

LOGICAL/SEMANTIC ERRORS

WHAT IS WRONG WITH THIS CODE?

```
1    def find_max_temperature(daily_temps:list)->float:
2        max_temp = 0
3        i = 0
4        while i < len(daily_temps):
5            if daily_temps[i] < max_temp:
6                max_temp = daily_temps[i]
7
8        return max_temp
```

How to fix logical/semantic errors

- Debugging: Use debugging tools and techniques to step through the code and inspect variable values
- Code reviews: Have peers review your code to catch logical mistakes
- Unit testing: Write tests to verify that your code produces the expected results
- Print statements: Use print statements to check intermediate values and ensure the logic is correct

PYTHON DEBUGGER

- TOOL THAT HELPS YOU TEST YOUR CODE BY ALLOWING YOU TO EXECUTE IT STEP-BY-STEP, INSPECT VARIABLES, AND CONTROL THE FLOW OF EXECUTION
- PYTHON PROVIDES THE PDB MODULE
- MOSTLY WORKS BY SETTING UP BREAKPOINTS IN THE CODE TO "PAUSE" THE EXECUTION

```
import pdb
pdb.set_trace()
# or
breakpoint() #Does not require importing pdb
```

Python Debugger – Basic Commands

- Navigating code
  - n (next): Execute the next line of code
  - c (continue): Continue execution until the next breakpoint
  - q (quit): Exit the debugger
- Inspecting variables
  - p variable_name: Print the value of a variable
  - l (list): Display the current line and surrounding code
- Controlling execution
  - s (step): Step into a function call
  - r (return): Continue execution until the current function returns

SIMPLE TESTS WITH ASSERT

```
10  #Unit test to check proper result
11  import random
12  positive_temps = list(range(4,34))
13  mix_temps = list(range(-20,34))
14  negative_temps = list(range(-20,-5))
15  random.shuffle(positive_temps)
16  random.shuffle(mix_temps)
17  random.shuffle(negative_temps)
18  assert find_max_temperature(positive_temps) == 33 #Test for all positive list
19  assert find_max_temperature(mix_temps) == 33 #Test for mixed (+ & -) list
20  assert find_max_temperature(negative_temps) == -6 #Test for all negative list
```

USING EXTERNAL RESOURCES

XML: Extensible Markup Language

- Markup language used to store and share data in a structured format
- Features:
  - Human readable: easy to read and understand
  - Hierarchical structure: Data is organized in a tree-like structure
  - Platform independent: Can be used across different systems and applications

XML FORMAT – IN GENERAL

- XML documents are formed as element trees, that start at a root element and branch into child elements
- XML elements can have attributes. These are used to contain data related to a specific element

```
<root>
    <child name="child1">Content of child1</child>
    <child name="child2">Content of child2</child>
</root>
```

XML FORMAT

```
<planets>
    <planet>
        <name>Mars</name>
        <mass>6.39E23</mass>
        <discovery>Unknown</discovery>
    </planet>
    <planet>
        <name>Jupiter</name>
        <mass>1.898E27</mass>
        <discovery>Unknown</discovery>
    </planet>
</planets>
```

Working with XML in Python

- Python provides several libraries for working with XML:
  - xml.etree.ElementTree
  - Lxml
  - Minidom

XML.ETREE.ELEMENTTREE

LET'S SEE HOW TO PARSE AN XML FILE AND ACCESSING ITS ELEMENTS BY EXPLORING OUR PLANETS FILE

```
import xml.etree.ElementTree as ET

tree = ET.parse('./data_files/planets.xml')
root = tree.getroot()
for planet in root:
    for planet_element in planet:
        print(planet_element.tag, planet_element.text)
    print("-"*50)
```

XML.ETREE.ELEMENTTREE

We can also find elements by their tag name

```
1    import xml.etree.ElementTree as ET
2
3    tree = ET.parse('./data_files/planets.xml')
4    root = tree.getroot()
5
6    for element in root.findall('.//name'):
7        print(element.text)
```

XML.ETREE.ELEMENTTREE

IT IS ALSO POSSIBLE TO WRITE OUR OWN XML FILES

```
import xml.etree.ElementTree as ET

root = ET.Element("cities")
city_1 = ET.SubElement(root, tag:"city")
name_c1 = ET.SubElement(city_1, tag:"name")
name_c1.text = "Toronto"
country_c1 = ET.SubElement(city_1, tag:"country")
country_c1.text = "Canada"

tree = ET.ElementTree(root)
tree.write("./data_files/cities.xml")
```

JSON: JavaScript Object Notation

- Lightweight data interchange format
  - easy for humans to read and write
  - easy for machines to parse and generate
- Language-independent: Can be used with many programming languages.
- Represents data as key-value pairs

JSON: JAVASCRIPT OBJECT NOTATION

```
{
    "planets": [
        {
            "name": "Saturn",
            "mass": 5.683E26,
            "discovery": "Unknown"
        }
    ]
}
```

JSON IN PYTHON

- Python provides the built-in JSON library for working with JSON data.
- Consider the following JSON file with data from multiple genes:


```
{
    "genes": [
        {
            "gene_id": "BRCA1",
            "name": "Breast Cancer 1",
            "chromosome": "17",
            "start_position": 43044295,
            "end_position": 43125482,
            "strand": "+",
            "function": "DNA repair",
            "associated_diseases": ["Breast cancer", "Ovarian cancer"]
        },
        {
            "gene_id": "TP53",
            "name": "Tumor Protein P53",
            "chromosome": "17",
            "start_position": 7668402,
            "end_position": 7687550,
            "strand": "+",
            "function": "Tumor suppression",
            "associated_diseases": ["Li-Fraumeni syndrome", "Various cancers"]
        },
```

JSON IN PYTHON

READING DATA FROM A JSON FILE LOADS A DICTIONARY

```
import json

with open("./data_files/genes.json", "r") as fh:
    data = json.load(fh) #Loads a dictionary
    genes = data["genes"] #Getting the list of genes
    for gene in genes:
        print(gene["gene_id"], gene["name"])
```

JSON IN PYTHON

READING DATA FROM A JSON FILE LOADS A DICTIONARY

```
import json

with open("./data_files/genes.json", "r") as fh:
    data = json.load(fh) #Loads a dictionary
    genes = data["genes"] #Getting the list of genes
    for gene in genes:
        print(gene["gene_id"], gene["name"])
```

JSON IN PYTHON

WE CAN ALSO WRITE JSON TO A FILE

```
import json

teams = {"MLB":["Toronto Blue Jays",
                "New York Mets",
                "Boston Red Sox"],
         "NBA":["Toronto Raptors",
                "Miami Heat",
                "Boston Celtics"]
         }

with open("./data_files/teams.json", "w") as out_file:
    json.dump(teams, out_file)
```

Web services and APIs

- Web services are standardized ways of integrating web-based applications using open standards over an Internet protocol backbone
- Two types:
  - **SOAP:** Uses XML for message format and relies on other application layer protocols like HTTP or SMTP.
  - **REST:** Uses HTTP requests to GET, PUT, POST, and DELETE data. Data is transferred in JSON format.

Web services and APIs

- An application programming interface (API) is a set of rules that allows software programs to communicate with each other.
- Main features:
  - **Endpoints:** Specific URLs where API services can be accessed
  - **Methods:** HTTP methods like GET, POST, PUT, DELETE are used to interact with the API

WEB SERVICES IN PYTHON

PYTHON PROVIDES THE REQUESTS LIBRARY FOR MAKING HTTP REQUESTS TO INTERACT WITH APIS

```
import requests

#URL to get the Toronto Blue Jays' roster
url = "https://statsapi.mlb.com/api/v1/teams/141/roster"

response = requests.get(url) #Call the API using the endpoint
data = response.json() #Get the JSON results
```

MLB stats API endpoints: https://github.com/toddrob99/MLB-StatsAPI/wiki/Endpoints

WEB SERVICES IN PYTHON

WE CAN ALSO USE MORE METHODS, E.G., PUT

```
import requests

url = "https://api.restful-api.dev/objects"

payload = {"name": "Apple MacBook Pro 16",
           "data": {
               "year": 2019,
               "price": 1849.99,
               "CPU model": "Intel Core i9",
               "Hard disk size": "1 TB"}
           }
response = requests.post(url, json=payload)
print(response.json())
print(response.status_code)
```

Web services in Python

We can use the status code to handle the responses properly. Some frequent codes include:

- 200 – OK
- 400 – Bad request
- 403 – forbidden
- 404 – Not found

```
import requests

url = "https://api.restful-api.dev/objects"

payload = {"name": "Apple MacBook Pro 16",
           "data": {
               "year": 2019,
               "price": 1849.99,
               "CPU model": "Intel Core i9",
               "Hard disk size": "1 TB"}
           }
response = requests.post(url, json=payload)
if response.status_code == 200:
    print("Request successful!")
else:
    print("Request failed")
```

Web services in Python

- Read the API's documentation
  - Understand the endpoints, methods used and required parameters
- Handle possible errors
  - Manage possible failed requests or unexpected responses using the status codes
- Secure API keys
  - Some APIs require a key to use them. Keep this key secure and avoid hardcoding them into your code.