Discovery 1: Parse API Data Formats with Python

Task 1: Prepare Development Environment

In this procedure, you will set up the environment for developing code in Python. First you will familiarize yourself with Visual Studio Code, which is an integrated development environment (IDE) where you will edit and manage the code with the included tools. You will review the files and folders in the working directory and try to run the Python code using pipenv, a tool for managing Python packages and virtual environments.

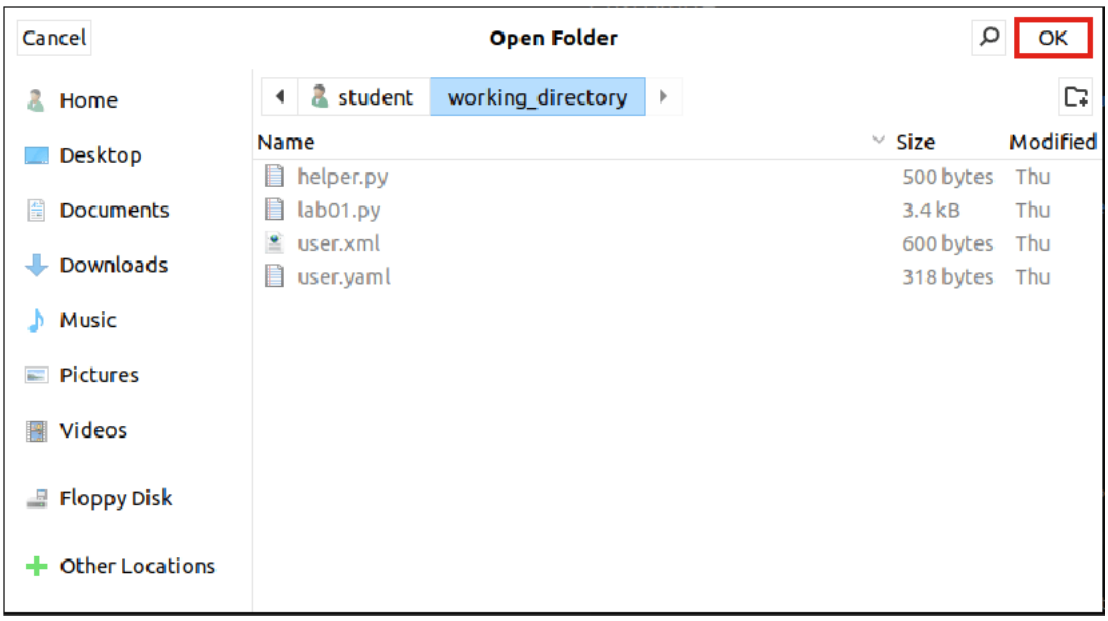
Virtual environments allow you to develop and run different Python projects independently, without having to worry about package version clashes (for example, an older project relying on an obsolete version of a package, while a different project requires the latest version). With a pipenv tool, you can select the version of Python your project will use, so some projects can still use the older version 2, while others use the newer version 3. You can also add or remove packages using pip inside the virtual environment; simply run **pipenv install** or **pipenv uninstall** instead of **pip install** or **pip uninstall**. The configuration of the virtual environment is kept inside the Pipfile to allow easy reinstallation if required.

Activity

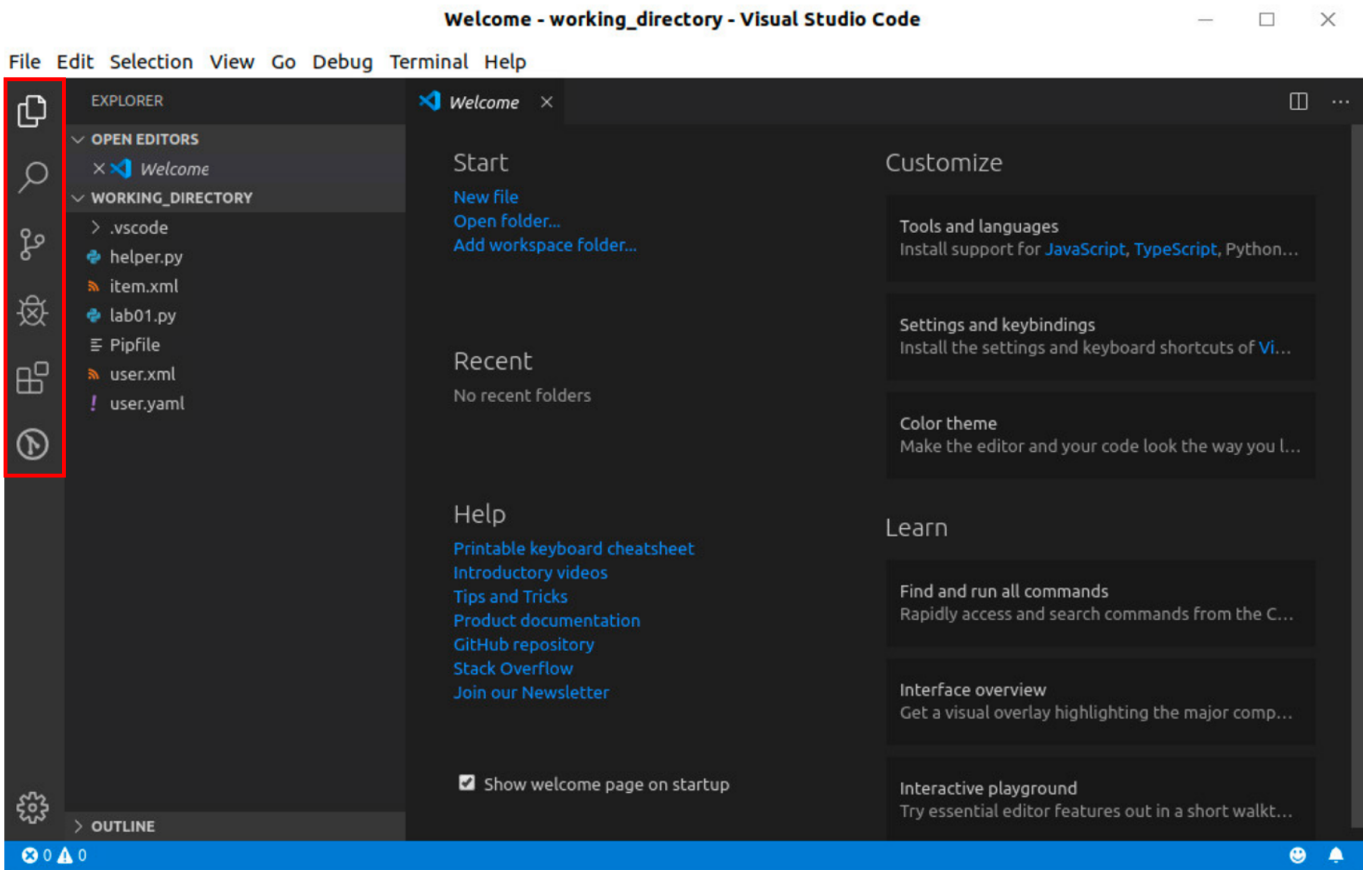
Step 1: From the desktop, open Visual Studio Code.

Step 2: Click **Open folder...** in the Start section on the Welcome page.

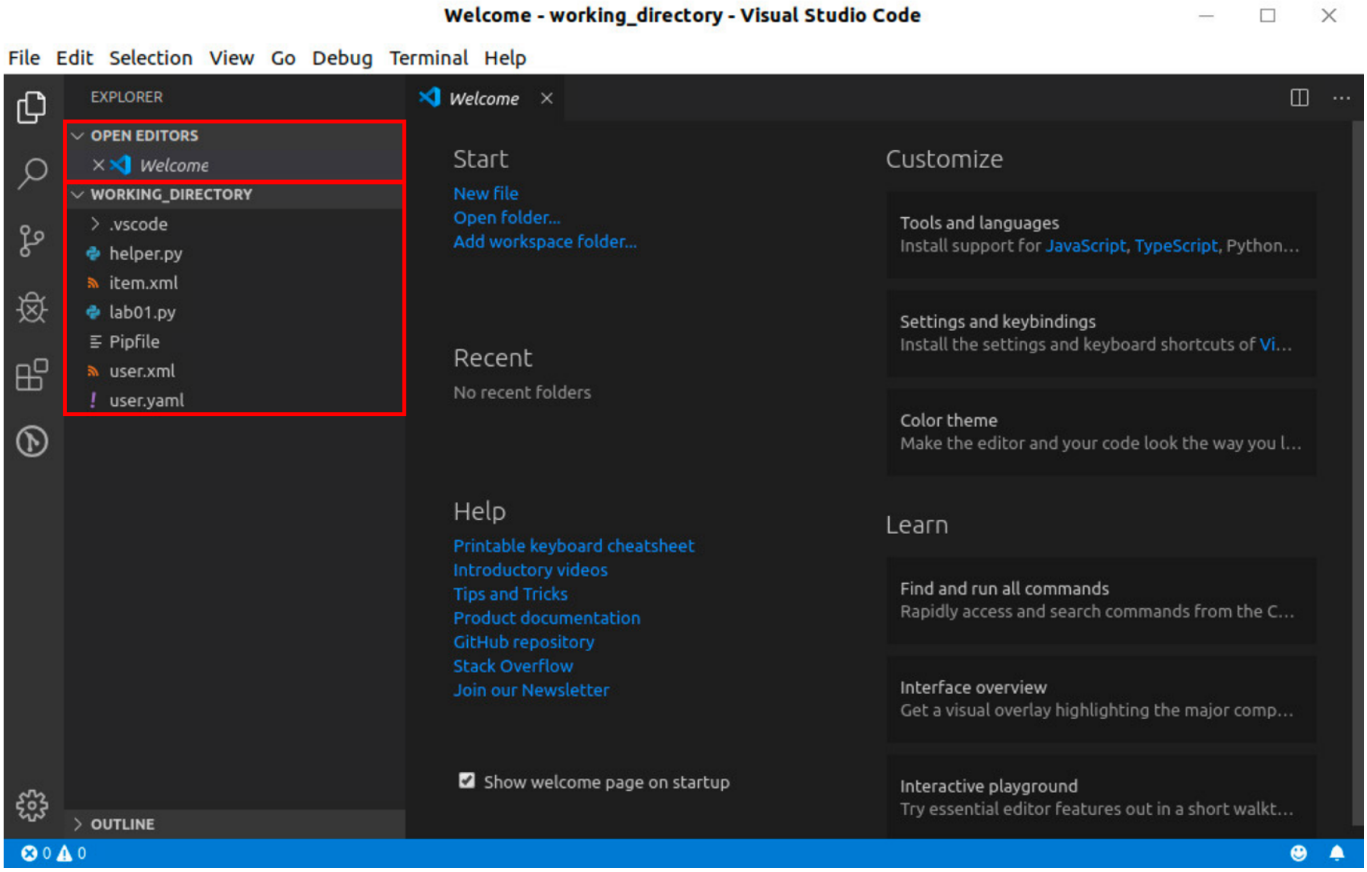
Step 3: To open the working environment, click **Home** and open **working_directory**. In the top-right corner, click **OK**.



By default, the activity bar on the left side consists of Explorer, Search, Source Control, Debug, Extensions and GitLens. Use **Explorer** to navigate the working directory.



Explorer consists of two sections. On the top is Open Editors, which lists the files that are currently open. The second part has the name of your working directory and shows its structure.



Step 4: There are already some files prepared in the working_directory on which you will be working. Open the Python file **lab01.py** by double-clicking it.

Note
The helper.py file contains the User class definition and some of the functions that are imported in lab01.py (from helper import *) and will be used in this lab.

By double-clicking a file, it is opened persistently, and it will not close when you open other files from Explorer.

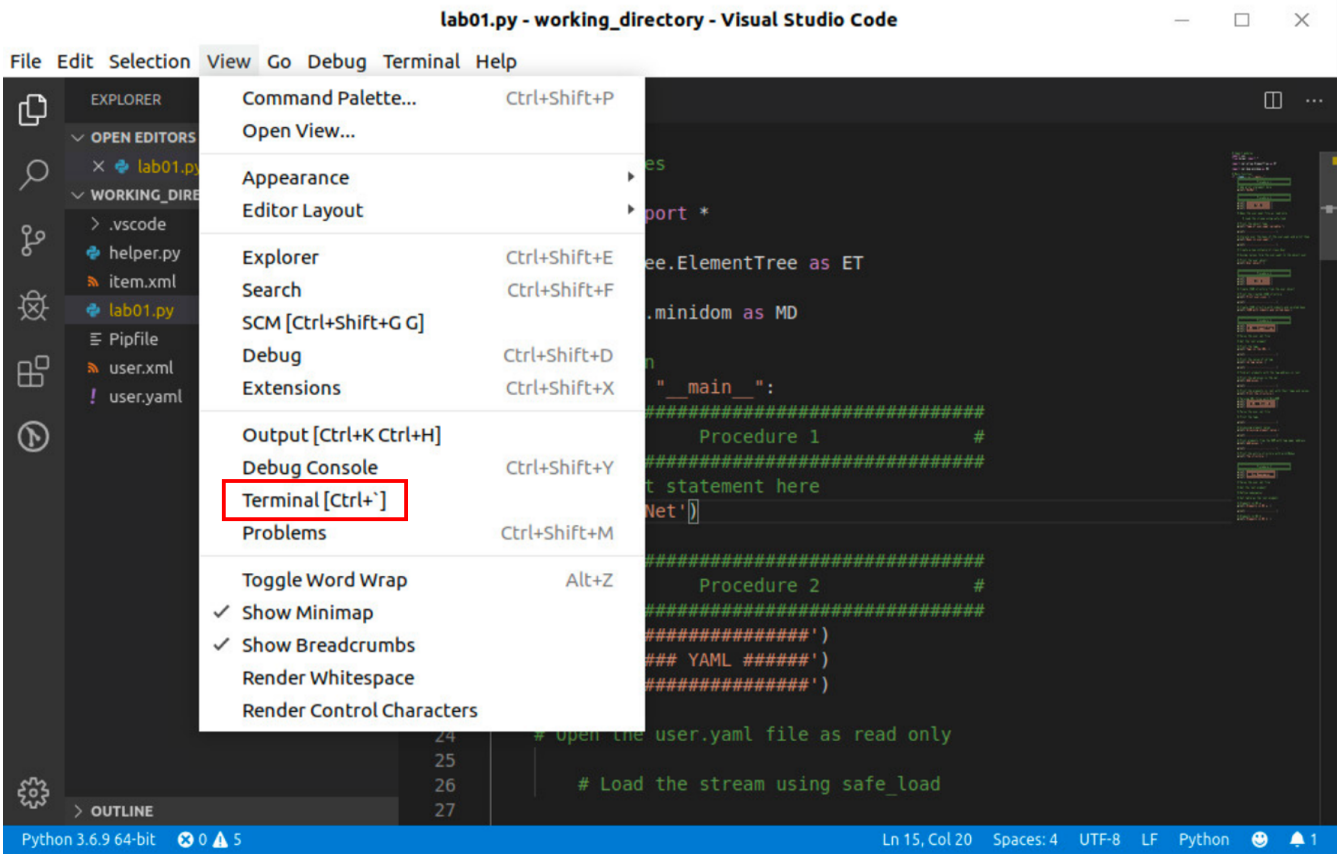
Step 5: In the main function, add a print statement that prints out "DevNet" below the comment "Add print statement here." Make sure to put spaces at the beginning of the line so that it lines up with the line above; indentation in Python is important.

```
if __name__ == "__main__":
    ########################################
    #              Procedure 1             #
    ########################################
    # Add print statement here
    print('DevNet')
```

Step 6: Save the modified file. Click **File > Save**, or use the shortcut **Ctrl + S**.

Remember to save the file every time you modify it.

Step 7: Before you run the code, add the terminal into the bottom panel. To use the terminal inside Visual Studio Code, click **View > Terminal** in the top menu. The terminal shows in the bottom panel. This way, you do not need to switch between applications to run your code.



Step 8: In the terminal, use the **pwd** command to check if you are in the directory /home/student/working_directory/.

```
student@student-workstation:~/working_directory$ pwd
/home/student/working_directory
```

Note
The '~' represents your home directory, the /home/student. In case you are not in the right directory, use the command **cd /home/student/working_directory** to move to the working directory.

Step 9: In the terminal, enter the **pipenv run python lab01.py** command to run the script.

```
student@student-workstation:~/working_directory$ pipenv run python lab01.py
<... output omitted ...>
DevNet
<... output omitted ...>
```

Note
The pipenv tool activates the virtual environment for you and runs the command inside. Because you have specified Python version 3 in the Pipfile, the **python** command will run that version, even if the system default version differs.

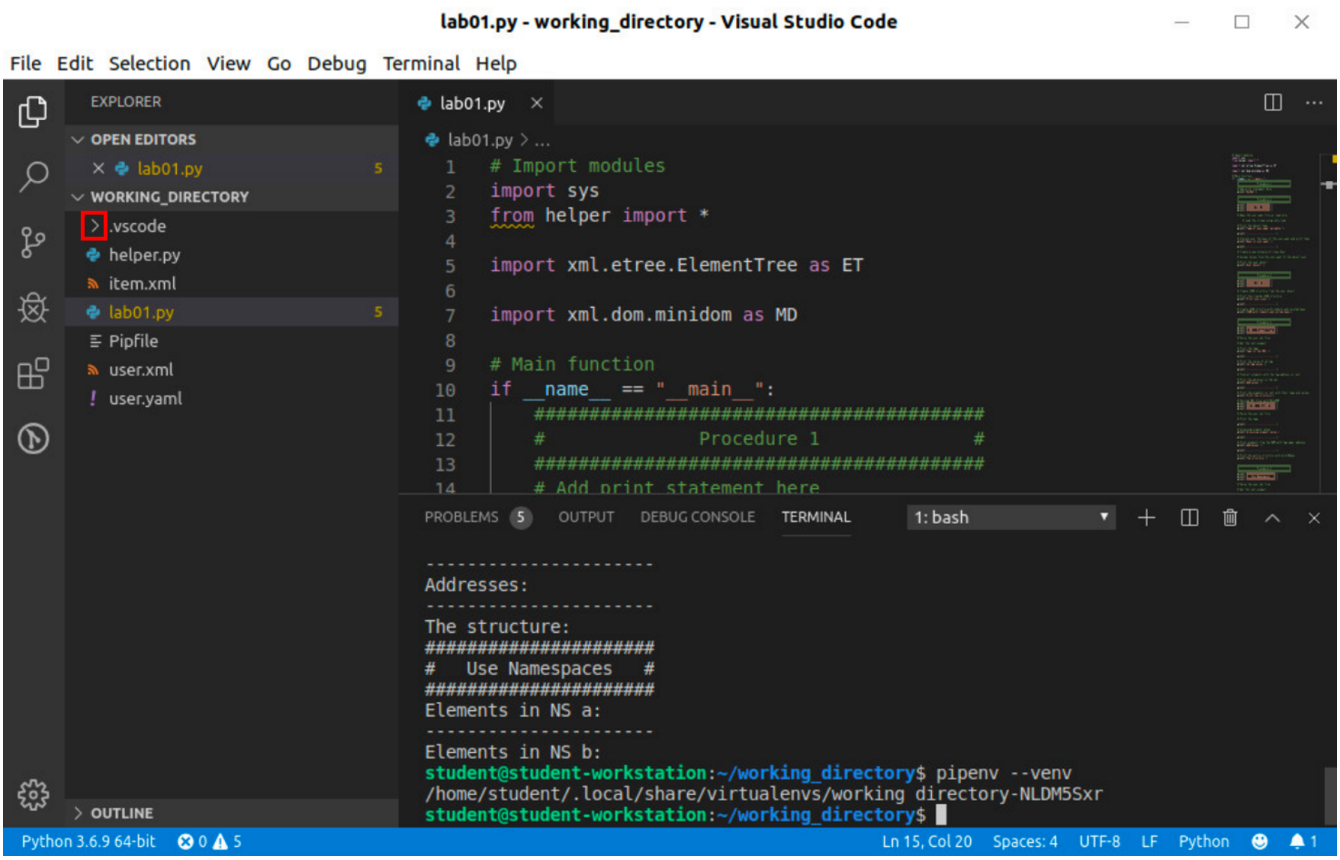
Step 10: For setting up pipenv, you need to get the path from where pipenv is executed. In the terminal, enter **pipenv --venv** and copy the path, by selecting it, right click on it and choose **Copy**.

Your path will be different from the following:

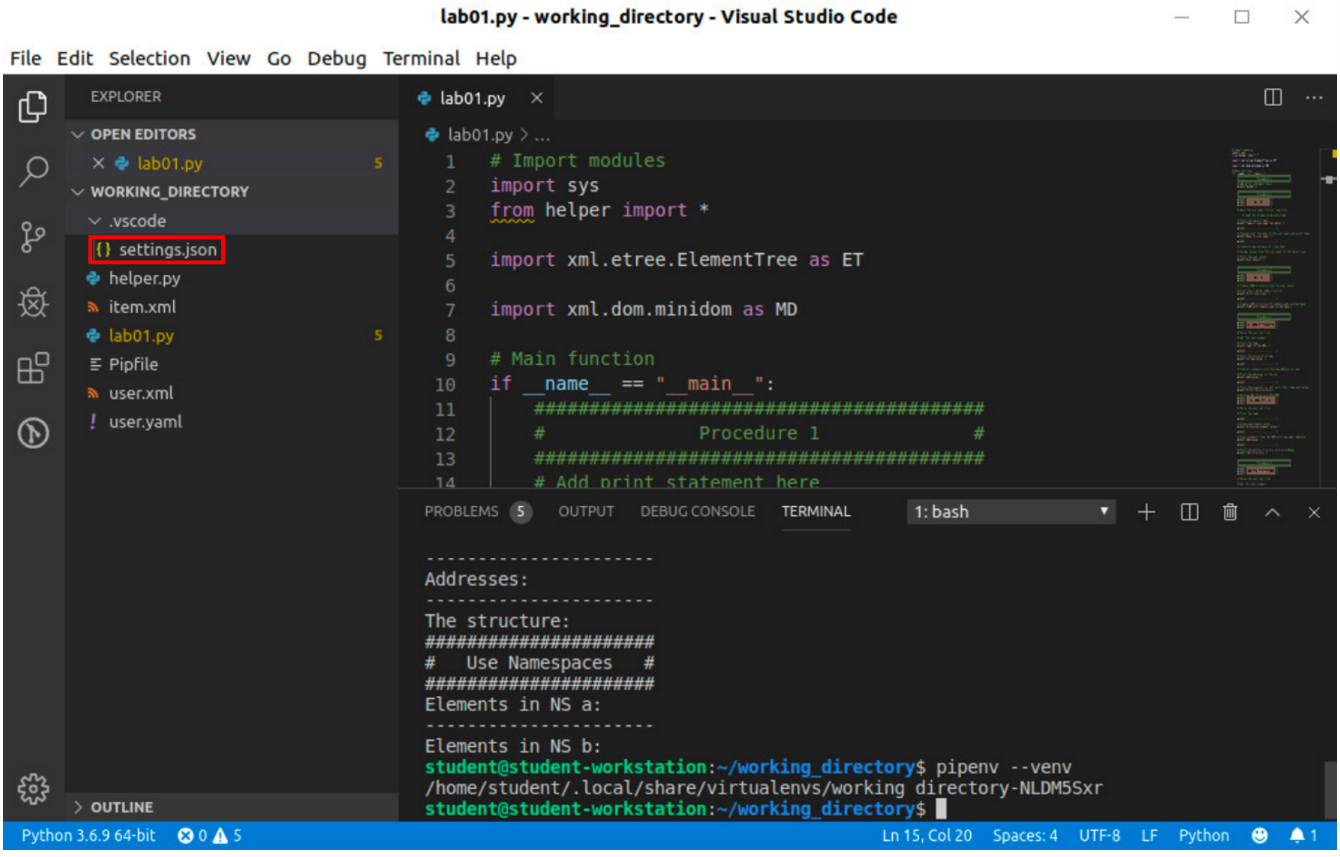
```
student@student-workstation:~$ pipenv --venv
/home/student/.local/share/virtualenvs/student-32m93jd
student@student-workstation:~$
```

Note
The path on your workstation may vary.

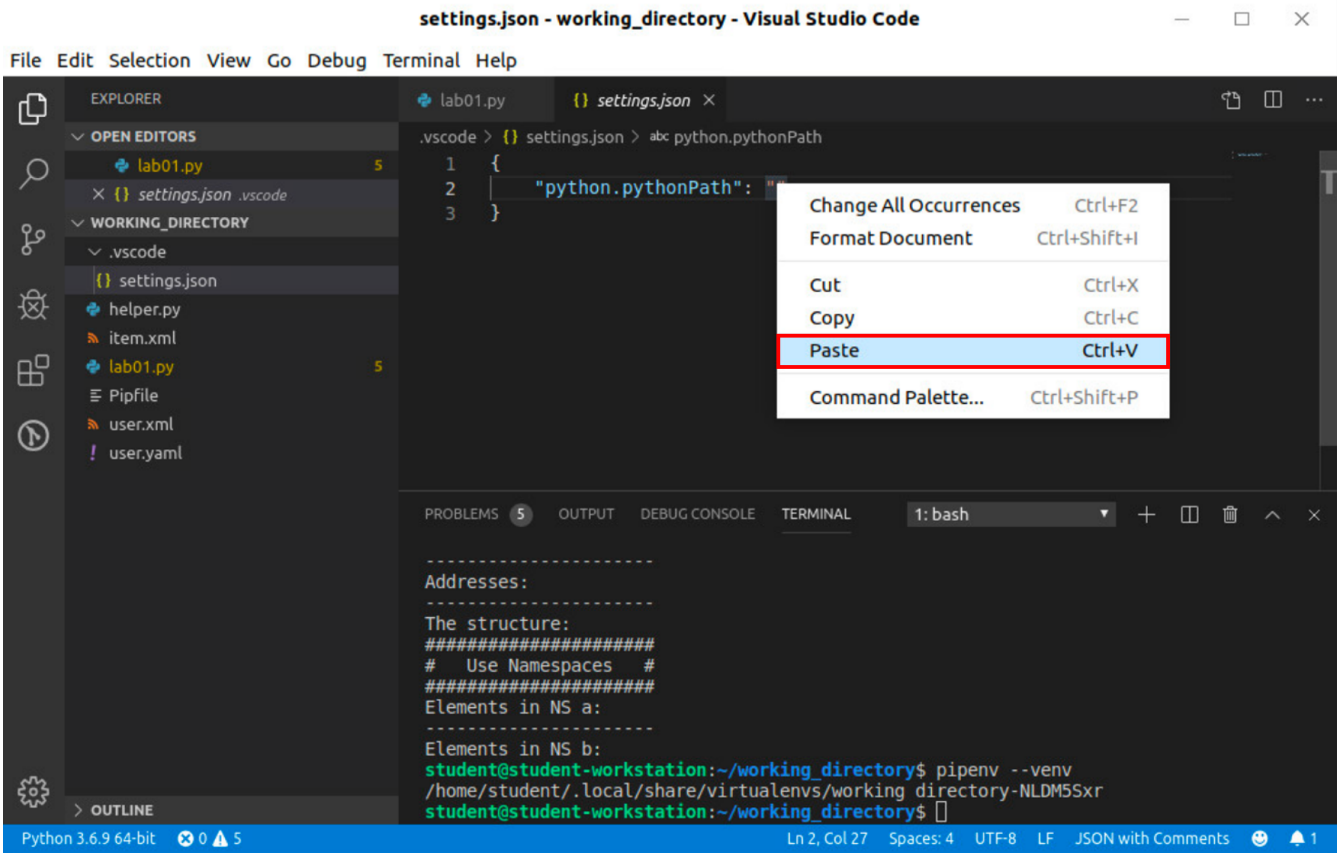
Step 11: In Explorer, expand the **.vscode** directory by clicking the arrow next to it.



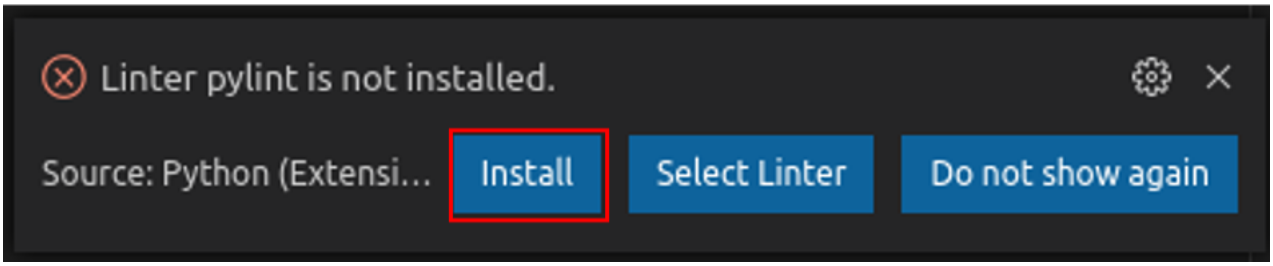
Step 12: Open the **settings.json** file in the .vscode directory.



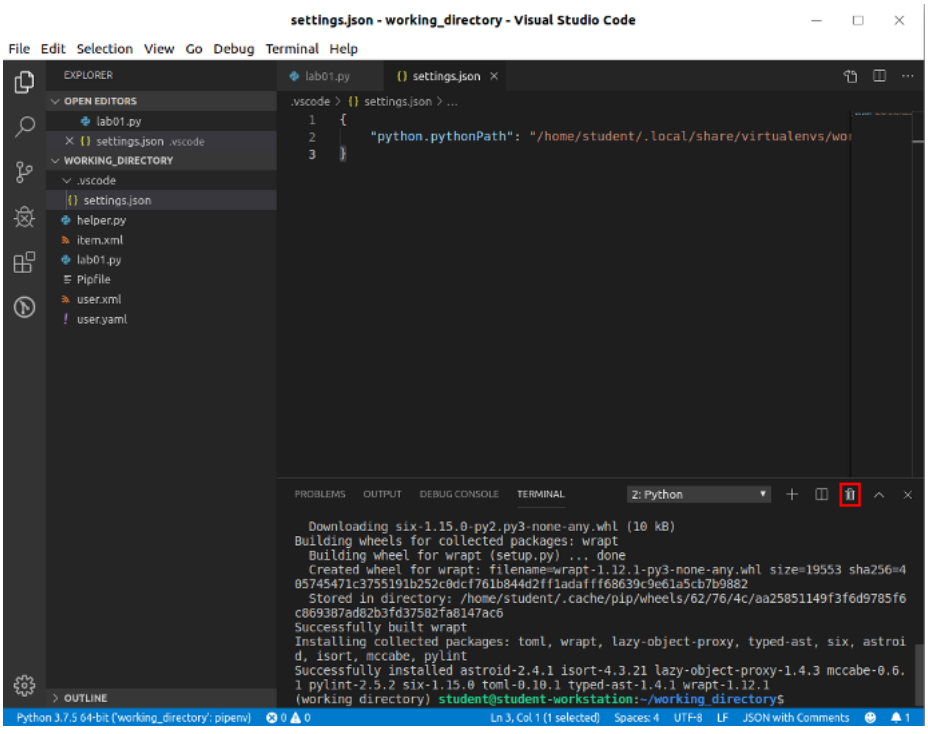
Step 13: In the empty double quotes, paste the previously copied path and **save** the file.



Step 14: A popup asking for intalling Linter pylint will show up. Click **Install** and wait for the installer to finish.



Step 15: Close the terminal used to install the Linter pylint by clicking the **Kill Terminal** icon.

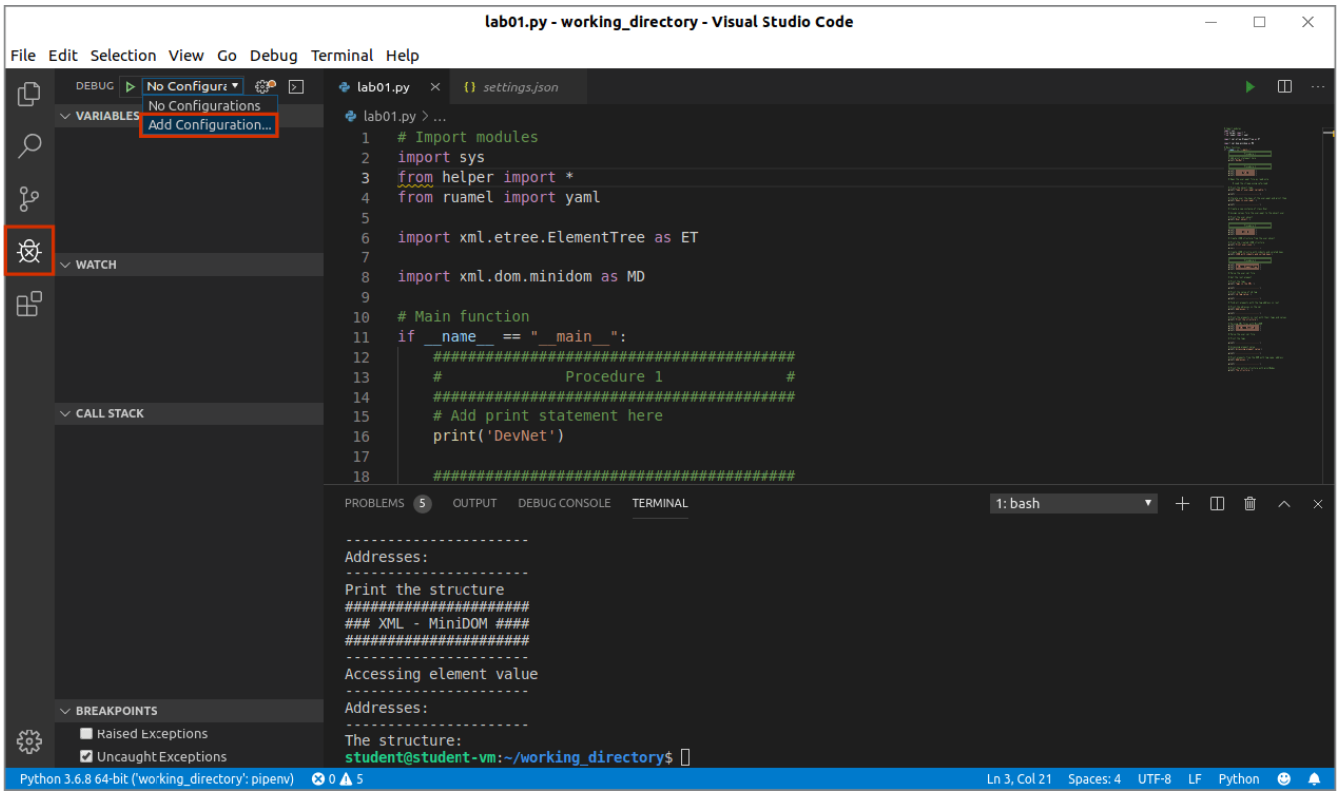


Step 16: Verify the path to the virtual environment in setting.json is in double quotes.

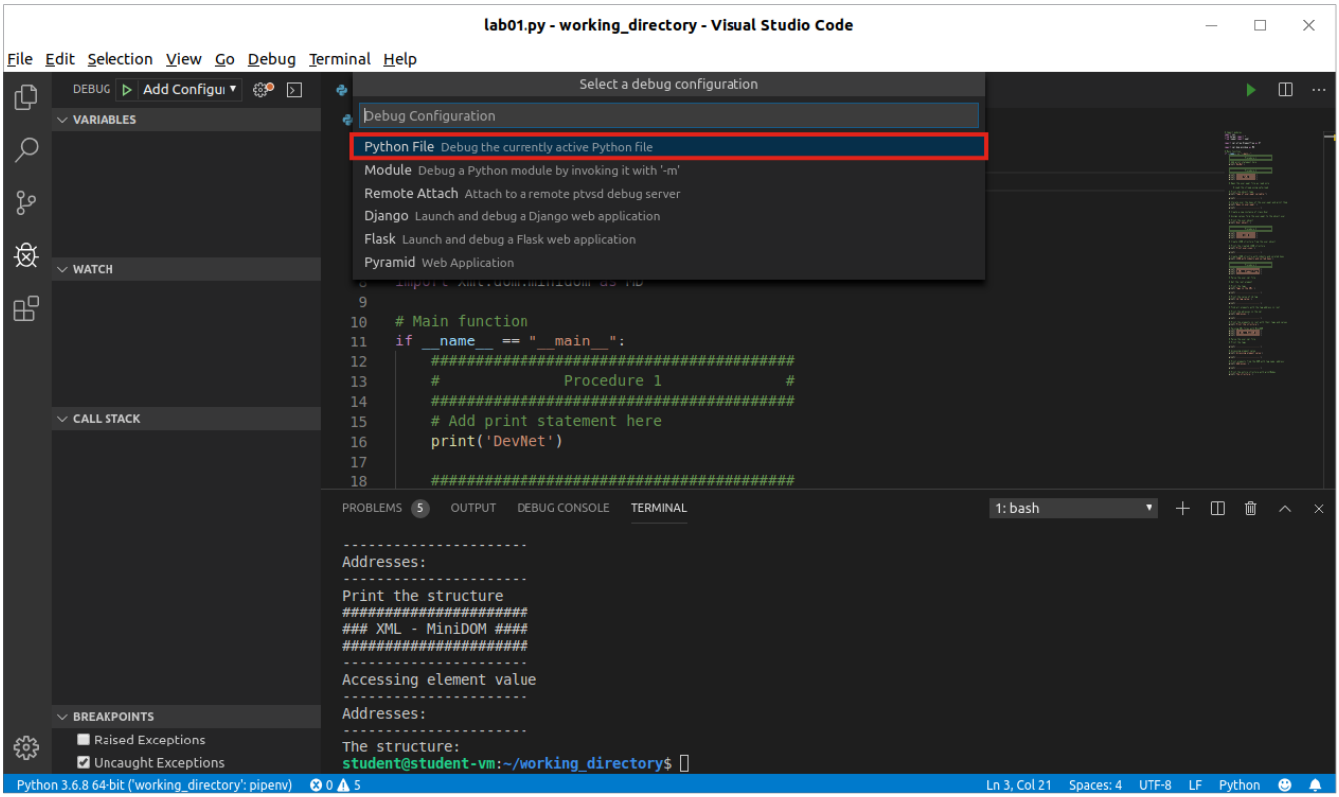
```
{
    "python.pythonPath": "/home/student/.local/share/virtualenvs/student-32m93jd"
}
```

Step 17: To run and debug the code, you can use the Debug Extension of Visual Studio Code. Before running the code for the first time using it, you need to set it up. First, make sure that the lab01.py file is open by clicking on its tab. Next, click the Debug Extension icon in the left part of the window. From the **No Configuration** drop-down menu that appears at the top of the window, select **Add Configuration...**.

Note
In order to be prompted for the Python interpreter that will be used, do not add the debug configuration from the Debug menu of the Visual Studio Code toolbar.

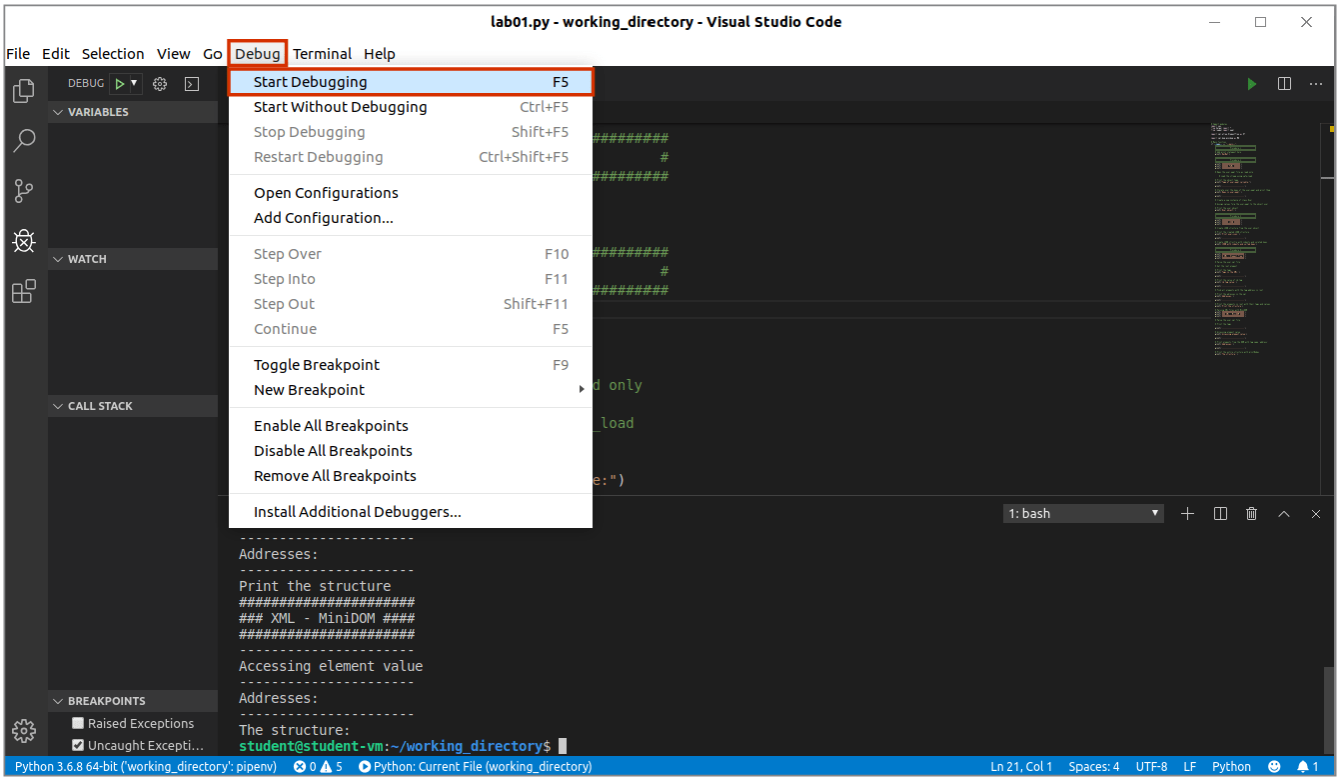


Step 18: Select the **Python File** option in the list to debug the currently open Python file.

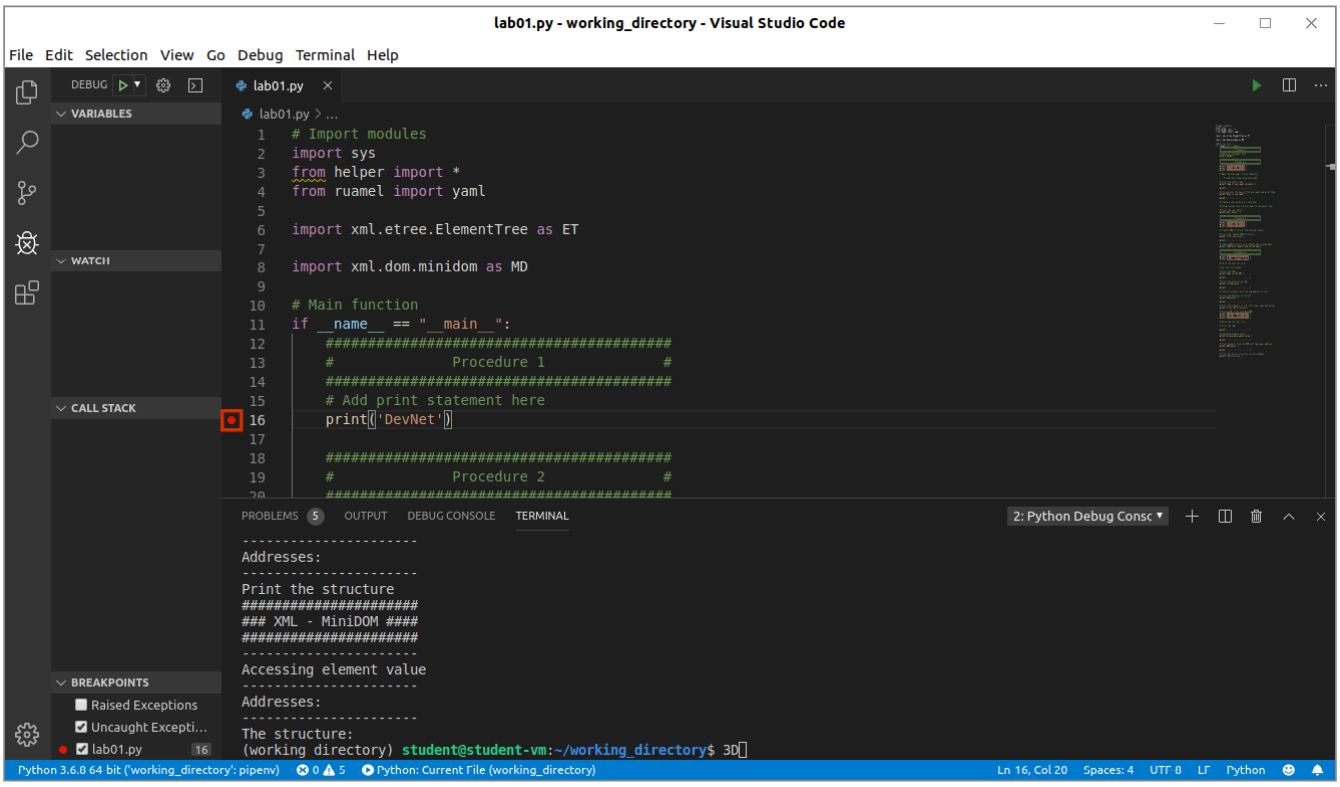


Step 19: The file launch.json opens with the configuration you have just created. Close the launch.json file and switch back to the lab01.py file.

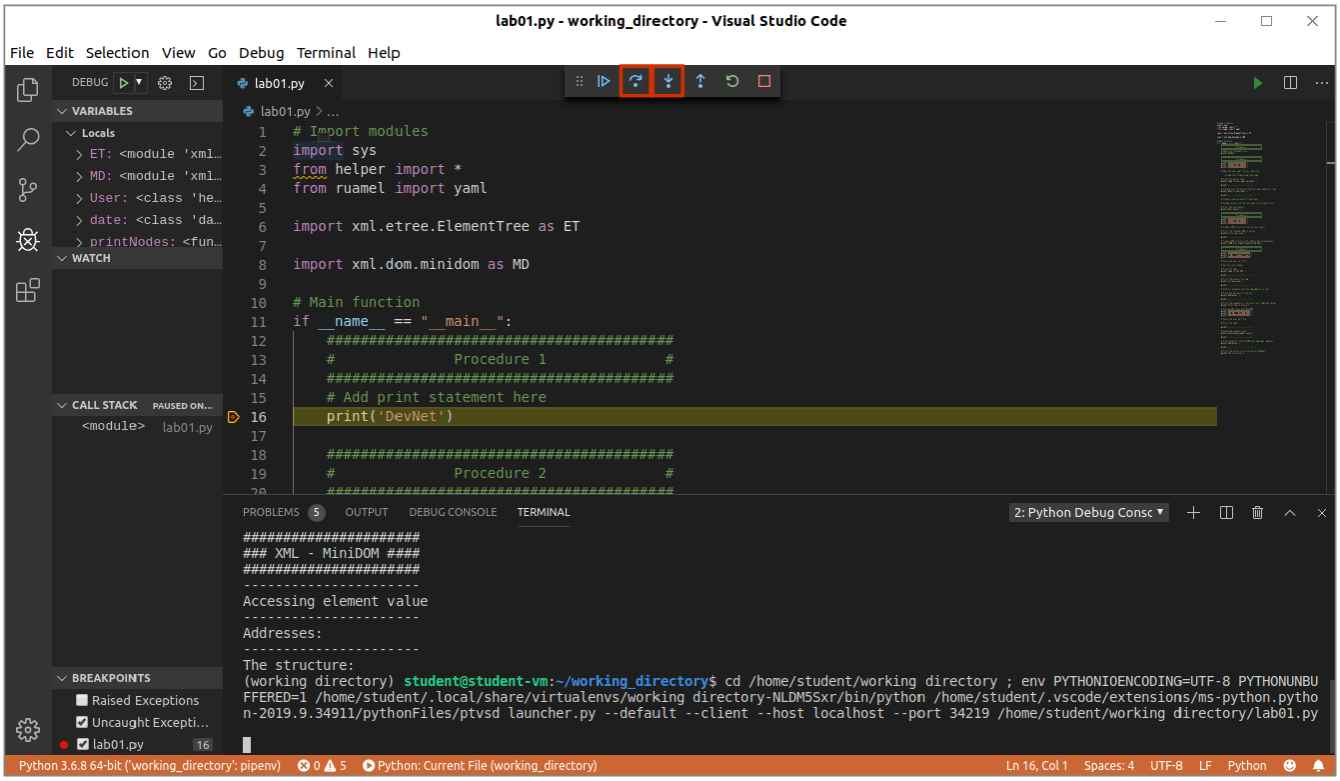
Step 20: From the top menu, choose **Debug > Start Debugging**. The shortcut you can use is function key F5. The code will run the same as before, in the terminal on the bottom.



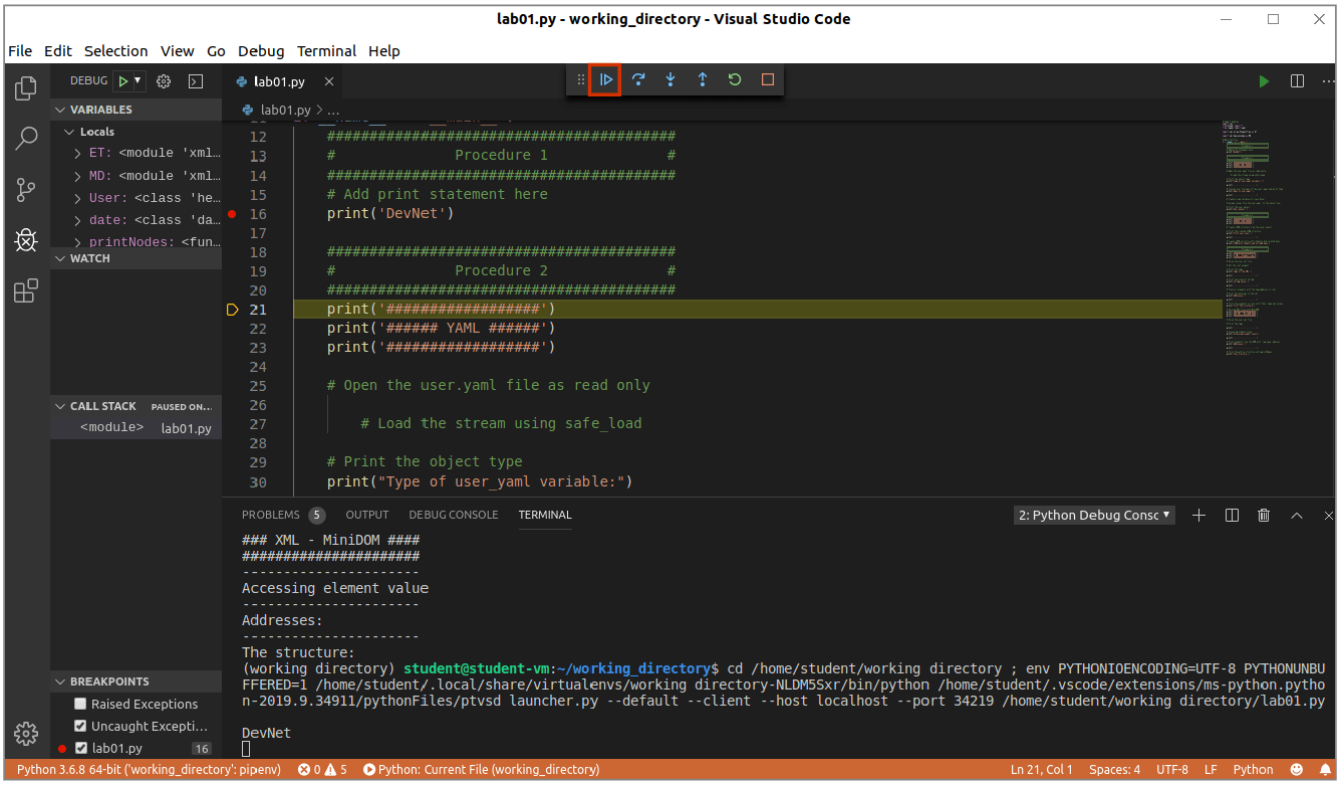
Step 21: For debugging (running the code step by step and checking its execution), add a breakpoint by clicking the red dot next to the number of the line where you print the "DevNet" string, and run the code by clicking **Debug > Start Debugging**.



Step 22: The program execution stops at the line where you have placed the breakpoint. On the left side, the Variables, Watch, and Call Stack show you the current state of the program running. On the top, the five icons are used to tell the debugger how to proceed with the execution of the code. The Play button runs the code to the next breakpoint (or to the end if there are none). The Step over button steps over a line. If the line is a function and you press **Step over**, the debugger will not debug each individual line of the function, while the next button, Step into, will. Step out tells the debugger to go back to the line that called the current function. The Restart button restarts the debugger, and the Stop button stops it. In this case, **Step over** and **Step into** are the same, so click either of the two to go to the next line.



Step 23: In the terminal, the "DevNet" string has been printed, and the execution of the code has moved to the next print statement. Click the **Play** icon on the top to run the rest of code.



Step 24: Remove the created breakpoint by clicking on it and leave the file lab01.py open.

Task 2: Parse YAML File

In this procedure, you will develop a Python script to parse YAML files using the ruamel.yaml module. The skeleton code with a user class and some functions are already prepared for you to use. You will learn how to use the ruamel.yaml module in Python and be able to parse the basic objects in YAML. In the lab01.py file, add the code underneath the appropriate comments.

Step 1: Ensure you have the **lab01.py** file open.

Step 2: The Python code is a skeleton with some of the necessary imports and with the main function. Read the comments to familiarize yourself with the structure before continuing. You will fill the missing code during the lab.

```
########################################
#               Procedure 2            #
########################################
print('##################')
print('###### YAML ######')
print('##################')
```

```
    # Open the user.yaml file as read only
```

Step 3: From the terminal, install the ruamel.yaml module using the **pipenv install** command. It is the equivalent of a pip install, except packages are installed inside your development environment, not systemwide.

```
(working directory) student@student-workstation:~/working_directory$ pipenv install ruamel.yaml
```

Note
Another popular module for manipulation of YAML files is PyYAML. The ruamel.yaml module is a derivative of the first one, is compatible with both Python version 2 and 3, and supports YAML 1.2, released in 2009.

Step 4: In the code, import the **ruamel.yaml** module. Place the import at the top of the file where the other imports are located.

```
# Import modules
import sys
from helper import *
from ruamel import yaml
```

Step 5: From the Explorer panel, open the **user.yaml** file and review how the file is structured. Keep the file open to review it when needed.

Step 6: Switch back to the **lab01.py** file. In the main function, use the **open** function to open user.yaml as a stream.

```
# Open the user.yaml file as read only
with open('user.yaml', 'r') as stream:
```

Step 7: Use the **safe_load** function from the ruamel.yaml module to load the stream. Save the object into a variable called **user_yaml**.

```
# Load the stream using safe_load
user_yaml = yaml.safe_load(stream)
```

Step 8: Determine which type of an object is the user_yaml variable, using the **type** function. Print the result.

```
# Print the object type
print("Type of user_yaml variable:")
print(type(user_yaml))
```

Step 9: Save the lab01.py file.

Step 10: Run the code and verify the type of object being displayed.

To run the code, in the menu at the top, click **Debug** > **Start Debugging**.

```
<... output omitted ...>
##################
###### YAML ######
##################
Type of user_yaml variable:
<class 'dict'>

<... output omitted ...>
```

Step 11: The user_yaml variable is a dictionary type of an object. Add the code to iterate over the keys of the user_yaml variable and print them, using the for loop. The dictionary keys are the same as in the user.yaml file, and you can use them to access the values in the user_yaml variable.

```
# Iterate over the keys of the user_yaml and print them
print('Keys in user_yaml:')
for key in user_yaml:
    print(key)
```

Step 12: Save and run the code. The keys in the dictionary are the same ones as in the user.yaml file.

```
<... output omitted ...>
Keys in user_yaml:
id
birth_date
score
first_name
last_name
address

<... output omitted ...>
```

Note
The order of the keys printed out might vary.

Step 13: Create a new instance of class User and name the variable **user**. The User object has multiple attributes that have already been defined in the User class, which is located in the helper.py file (helper module). You will assign the values from the user_yaml variable into this object.

```
# Create a new instance of class User
user = User()
```

Step 14: Assign the user object attributes the values from the corresponding user_yaml keys. The object attributes and the dictionary keys have the same names. You can also check the attribute names in the helper.py file.

```
# Assign values from the user_yaml to the object user
user.id = user_yaml['id']
user.first_name = user_yaml['first_name']
user.last_name = user_yaml['last_name']
user.birth_date = user_yaml['birth_date']
user.address = user_yaml['address']
user.score = user_yaml['score']
```

Step 15: Print the **user** object.

```
# Print the user object
print('User object:')
print(user)
```

Step 16: Save and run the code. The user object is printed in a dictionary format.

```
<... output omitted ...>
User object:
{'score': 18.3, 'id': 3242, 'last_name': 'Smith', 'first_name': 'Ray', 'birth_date': datetime.date(1979,
<... output omitted ...>
```

Note
The order of the keys and their corresponding values printed out might vary.

Task 3: Write and Inspect JSON File

In this procedure, you will continue working on the code from the previous procedure. You will prepare the created user object to be JSON serializable and create the JSON structure using the **dumps** function from the json package.

Step 1: At the top of the lab01.py file, import the **json** module.

```
# Import modules
import sys
from helper import *
from ruamel import yaml
import json
```

Step 2: The **dumps** function takes a variable and produces a JSON string. It knows how to encode dictionaries, lists, and simple values, such as strings and numbers. However, custom objects, like the *user* variable, might have internal, private state that makes no sense if written to a string. That's why user-defined objects require a custom serialization function; specify it as a named-argument called default. Use the provided **serializeUser** function from the helper module to serialize the user variable. Save the JSON structure into the *user_json* variable.

```
# Create JSON structure from the user object
user_json = json.dumps(user, default = serializeUser)
```

Note
The full documentation of the json module is available at .

Step 3: Add a statement to print the **user_json** variable.

```
# Print the created JSON structure
print('Print user_json:')
print(user_json)
```

Step 4: Save and run the code to verify the JSON structure.

```
<... output omitted ...>
Print user_json:
{"birth_date": "1979-08-15", "first_name": "Ray", "id": 3242, "score": 18.3, "last_name": "Smith", "addre
<... output omitted ...>
```

Note
The order of the keys and their corresponding values printed out might vary.

Step 5: In Visual Studio Code, in the editor, explore the additional arguments of the **dumps** function by hovering the cursor over it.

Step 6: Output the JSON structure in structured, readable form with the keys in alphabetical order by setting the sort_keys parameter to **true**. Set the parameter indent to **4** in the **dumps** method.

```
# Create JSON structure with indents and sorted keys
print('JSON with indents and sorted keys')
user_json = json.dumps(user, default = serializeUser, indent=4, sort_keys=True)
print(user_json)
```

Step 7: Save and run the code to see how the JSON structure looks.

```
<... output omitted ...>
JSON with indents and sorted keys
{
    "address": [
        {
            "city": "Royal Oak",
            "postal_code": 44663,
            "primary": 1,
            "state": "OH",
            "street": "94873 Ledner Rue"
        },
        {
            "city": "Elnaville",
            "postal_code": 17319,
            "primary": 0,
            "state": "EL",
            "street": "832 William Ave"
        }
    ],
    "birth_date": "1979-08-15",
    "first_name": "Ray",
    "id": 3242,
    "last_name": "Smith",
    "score": 18.3
}
<... output omitted ...>
```

Task 4: Compare Different XML Parsers

In this procedure, you will use two different XML parsers, ElementTree and MiniDOM, to parse an XML file and achieve the same result. You will learn how each of the two work, what are their differences, and you will be able to determine in which cases to use one or the other.

Step 1: From Explorer, open the **user.xml** file and review it. The XML file contains the same information as the YAML file on which you have been working.

Step 2: Switch back to the **lab01.py** file and scroll to the top. The ElementTree module has been imported as ET.

```
import xml.etree.ElementTree as ET
```

Step 3: In the "XML - Element Tree" section of the script, use the parse function of ET to parse the user.xml file. Save the ElementTree instance into a variable named **tree**.

```
########################################
#              Procedure 4             #
########################################
print('######################')
print('# XML - Element Tree #')
print('######################')

# Parse the user.xml file
tree = ET.parse('user.xml')
```

Step 4: Use the **getroot()** function of ElementTree to get the root element of the tree variable and save it into a variable named *root*.

```
# Get the root element
root = tree.getroot()
```

Step 5: Print the tags of the XML file using a for loop to iterate through the elements in the root variable. The tag is accessible with the .tag attribute of an element.

```
# Print the tags
print('Tags in the XML:')
for element in root:
    print(element.tag)
```

Step 6: Save and run the code to check the tags.

```
<... output omitted ...>
Tags in the XML:
id
first_name
last_name
birth_date
address
address
score
<... output omitted ...>
```

Step 7: The **find** function of the Element object takes a string as a parameter and returns the Element with the same tag name as the string. Use the **find** function on the root variable to find the ID element, and use the .text attribute of the Element object to get the value of the ID tag and print it.

```
# Print the value of id tag
print('id tag value:')
print(root.find('id').text)
```

Step 8: Save and run the code to verify the value of the ID tag.

```
<... output omitted ...>
id tag value:
3242
<... output omitted ...>
```

Step 9: Similar to the **find** function, which finds the first element with the specified tag, the **findall** function returns all the elements with the specified tag. Use the **findall** function of the root variable to get all elements with the tag "address", and assign the value to a variable named **addresses**.

```
# Find all elements with the tag address in root
addresses = root.findall('address')
```

Step 10: Write a nested loop to print the tags and values of the XML elements in the addresses variable.

```
# Print the addresses in the xml
print('Addresses:')
for address in addresses:
    for i in address:
        print(i.tag + ':' + i.text)
```

Step 11: Run the code. The two addresses with their corresponding values are printed.

```
<... output omitted ...>
Addresses:
street:94873 Ledner Rue
city:Royal Oak
postal_code:44663
state:OH
primary:1
street:832 William Ave
city:Elnaville
postal_code:17319
state:EL
primary:0
<... output omitted ...>
```

Step 12: Iterate through the root variable using the **iter** function and print all element tags and values.

```
# Print the elements in root with their tags and values
print('Print the structure')
for k in root.iter():
    print(k.tag + ':' + k.text)
```

Step 13: Save and run the code. The entire ElementTree structure is printed in a readable way.

```
<... output omitted ...>
user:

id:3242
first_name:Ray
last_name:Smith
birth_date:1979-08-15
address:

street:94873 Ledner Rue
city:Royal Oak
postal_code:44663
state:OH
primary:1
address:

street:832 William Ave
city:Elnaville
postal_code:17319
state:EL
primary:0
score:18.3
<... output omitted ...>
```

Step 14: The MiniDOM module has been imported at the top of the file as MD, which you will use in the next steps.

```
import xml.dom.minidom as MD
```

Step 15: In the MiniDOM section, add the statement to parse the user.xml file, using the **parse** function of the MD module. Save the returned DOM object into a variable named **dom**.

```
# Parse the user.xml file
dom = MD.parse('user.xml')
```

Step 16: Use the **printTags** function in the helper module to print the tags of the child nodes of the dom variable.

Iterate through the child nodes of the dom variable, and pass the child nodes list of the iterator to the **printTags** function.

```
# Print the tags
print('Tags in the XML:')
for node in dom.childNodes:
    printTags(node.childNodes)
```

Note
The **printTags** function in the helper module iterates through the list of nodes of the parameter and prints the node name (tag).

Step 17: Save and run the code to print the tags.

```
<... output omitted ...>
Tags in the XML:
id
first_name
last_name
birth_date
address
address
score
<... output omitted ...>
```

Step 18: In the section "Accessing element value", use the **getElementsByTagName** function and pass the "id" tag as a string for the argument. Assign the returned value to the **idElements** variable. The variable will contain a list of NodeList objects with nodes that have the tag name "id".

```
# Accessing element value
print('Accessing element value')
idElements = dom.getElementsByTagName('id')
print (idElements)
```

Step 19: Save and run the code.

```
<... output omitted ...>
Accessing element value
[<DOM Element: id at 0x7f5ca6df50e0>]
<... output omitted ...>
```

Note
The location of the DOM Element in the memory differs each time the code is run, so your output will differ from the example.

Step 20: To access the Element object in a NodeList object, use the **item** function and pass it an integer value. Because there is a single item in the idElements node list, the integer value should be 0. Save the Element object into a variable named **elementId**.

```
# Accessing element value
print('Accessing element value')
idElements = dom.getElementsByTagName('id')
print (idElements)
elementId = idElements.item(0)
```

Step 21: Print the child nodes of the elementId element using the property childNodes, which returns a list of nodes contained within elementId.

```
print(elementId.childNodes)
```

Step 22: Save and run the code to check the child nodes of the elementId variable.

```
<... output omitted ...>
Accessing element value
[<DOM Element: id at 0x7f47f42350e0>]
[<DOM Text node "'3242'">]
<... output omitted ...>
```

Step 23: Because there is a single element in the elementId node, use the firstChild property to access it and the data property to get the value of the ID tag. Save the value into a variable named **idValue**.

```
idValue = elementId.firstChild.data
```

Step 24: Print the **idValue** variable to verify that the obtained value is the correct one.

```
print(idValue)
```

Step 25: Save and run the code to verify the value of the ID tag.

```
<... output omitted ...>
Accessing element value
[<DOM Element: id at 0x7f47f42350e0>]
[<DOM Text node "'3242'">]
3242
<... output omitted ...>
```

Step 26: Use the functions **getElementsByTagName** and **printNodes** to print all the child elements of the address tags. Loop over the elements and pass the child nodes to the **printNodes** function from the helper.py module.

```
# Print elements from the DOM with tag name 'address'
print('Addresses:')
for node in dom.getElementsByTagName('address'):
    printNodes(node.childNodes)
```

Step 27: Save and run the code. The addresses are the same as in the XML file.

```
<... output omitted ...>
Addresses:
street:94873 Ledner Rue
city:Royal Oak
postal_code:44663
state:OH
primary:1
street:832 William Ave
city:Elnaville
postal_code:17319
state:EL
primary:0
<... output omitted ...>
```

Step 28: In a similar way as before, write the code that prints all the nodes and their values of the dom variable. Iterate over the child nodes of the dom variable, and pass the child nodes for the iterator to the **printNodes** function.

```
# Print the entire structure with printNodes
print('The structure:')
for node in dom.childNodes:
    printNodes(node.childNodes)
```

Step 29: Save and run the code to verify that the structure is correct.

```
<... output omitted ...>
The structure:
id:3242
first_name:Ray
last_name:Smith
birth_date:1979-08-15
address:

  street:94873 Ledner Rue
  city:Royal Oak
  postal_code:44663
  state:OH
  primary:1
address:

  street:832 William Ave
  city:Elnaville
  postal_code:17319
  state:EL
  primary:0
score:18.3
<... output omitted ...>
```

Task 5: Use Namespaces

In this procedure, you will parse an XML file that has the same tag names but in two different namespaces.

Activity

Step 1: From Explorer in Visual Studio Code, open the **item.xml** file. There are two table tags, belonging to different namespaces.

```
<?xml version="1.0"?>
<item>
    <a:table xmlns:a="https://www.example.com/network">
        <a:tr>
            <a:td>Router</a:td>
```

```
            <a:td>Switch</a:td>
        </a:tr>
    </a:table>
    <b:table xmlns:b="https://www.example.com/furniture">
        <b:name>Coffee Table</b:name>
        <b:length>180</b:length>
        <b:width>80</b:width>
    </b:table>
</item>
```

Step 2: In the "Use Namespace" section of the lab01.py file, use the **parse** function of ET to parse the item.xml file. Save the ElementTree instance into a variable named **itemTree**.

```
print('######################')
print('#   Use Namespaces   #')
print('######################')

# Parse the user.xml file
itemTree = ET.parse('item.xml')
```

Step 3: Use the **getroot** function of ElementTree to get the root element of the itemTree variable.

```
# Get the root element
root = itemTree.getroot()
```

Step 4: Define the two namespaces into a dictionary named **namespaces**. The keys are 'a' and 'b', and the corresponding values are the a and b namespaces from the item.xml file.

```
# Define namespaces
namespaces = {'a':'https://www.example.com/network', 'b':'https://www.example.com/furniture'}
```

Step 5: Using the **findall** function, set the table element as the root element of the two namespaces. Use **elementsInNSa** as the variable for the 'a' namespace and **elementsInNSb** for the 'b' namespace. Pass the **findall** function the tags and the namespaces dictionary.

```
# Set table as the root element
elementsInNSa = root.findall('a:table', namespaces)
elementsInNSb = root.findall('b:table', namespaces)
```

Step 6: Use a double for loop to print the tags and texts of the elements in the namespace 'a'.

```
# Elements in NS a
print('Elements in NS a:')
for e in elementsInNSa:
    for i in e.iter():
        print(i.tag + ':' + i.text)
```

Step 7: Save and run the code to print the elements in the namespace 'a'.

```
<... output omitted ...>
Elements in NS a:
{https://www.example.com/network}table:

{https://www.example.com/network}tr:

{https://www.example.com/network}td:Router
{https://www.example.com/network}td:Switch
<... output omitted ...>
```

Step 8: Similarly, do the same for the namespace 'b'. Use the **list** function; pass the first element of elementsInNSb and iterate over these elements. Print the tag and text of each element.

```
# Elements in NS b
print('Elements in NS b:')
for element in list(elementsInNSb[0]):
    print(element.tag + ':' + element.text)
```

Step 9: Save and run the code to print the elements in the namespace 'b'.

```
<... output omitted ...>
Elements in NS b:
{https://www.example.com/furniture}name:Coffee Table
{https://www.example.com/furniture}length:180
{https://www.example.com/furniture}width:80
<... output omitted ...>
```