Instructor:

Samyan Qayyum Wahla

Objective

Configuring the Environment

We will be using spyder throughout this course for the completion of assignments, the following steps will take you through the step the step instructions for the installation of the IDE.

Download the IDE:

Download miniconda from https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe .

Installation

Figure 1 Step 1 for IDE Installation

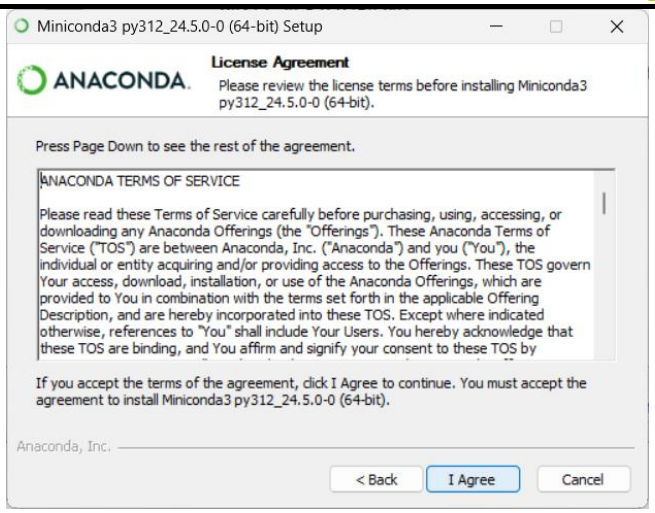


Figure 2 Step 2 for IDE Installation

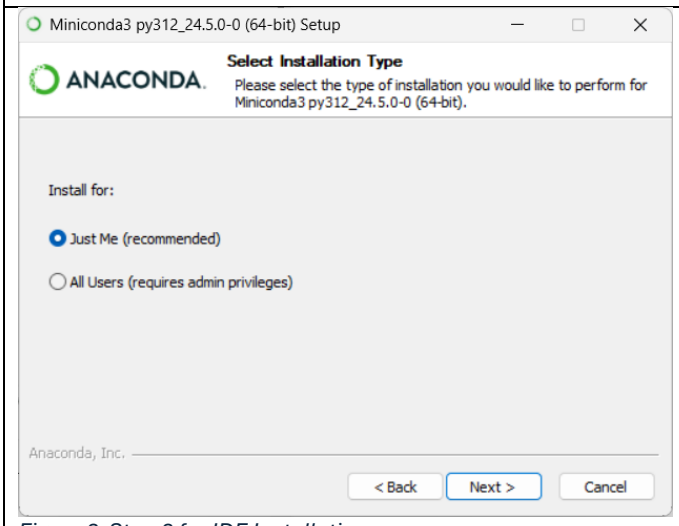


Figure 3 Step 3 for IDE Installation

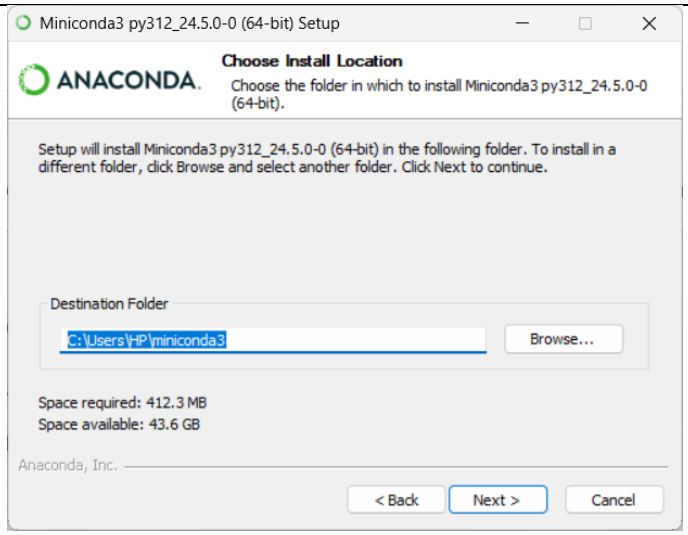


Figure 4 Step 4 for IDE Installation

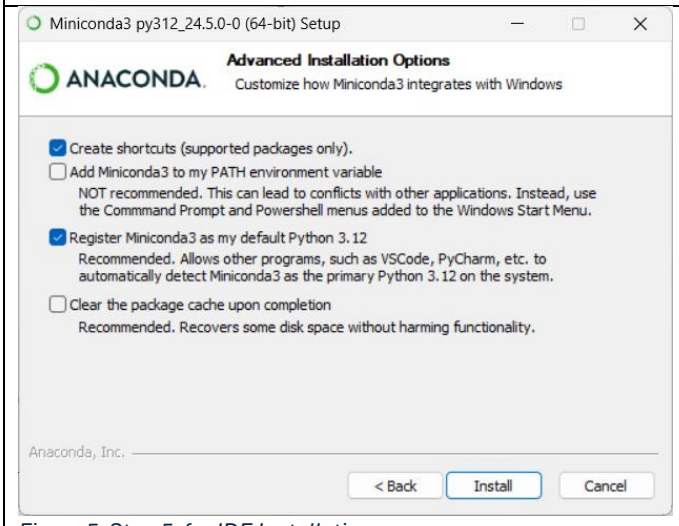


Figure 5 Step 5 for IDE Installation

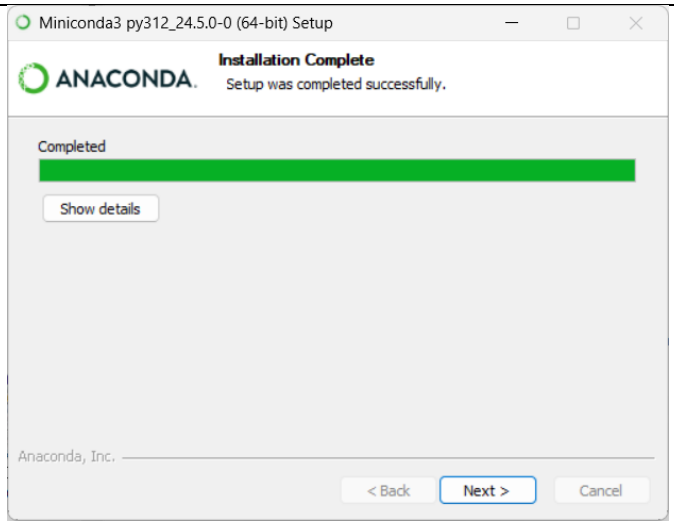


Figure 6 Step 6 for IDE Installation

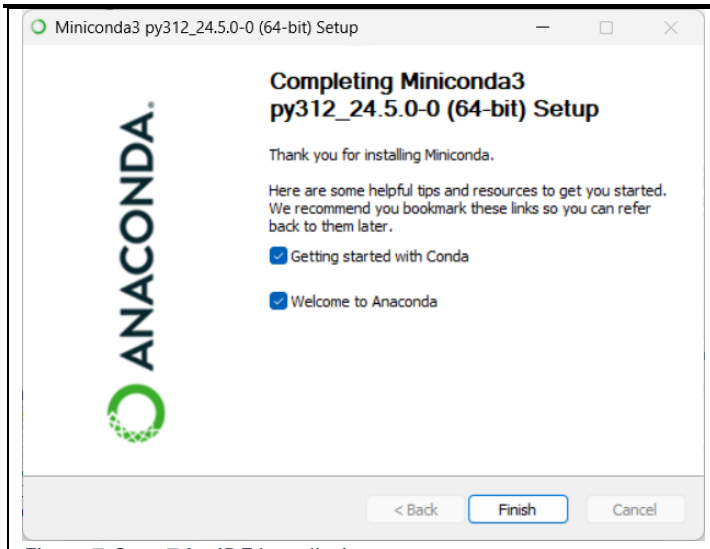


Figure 7 Step 7 for IDE Installation

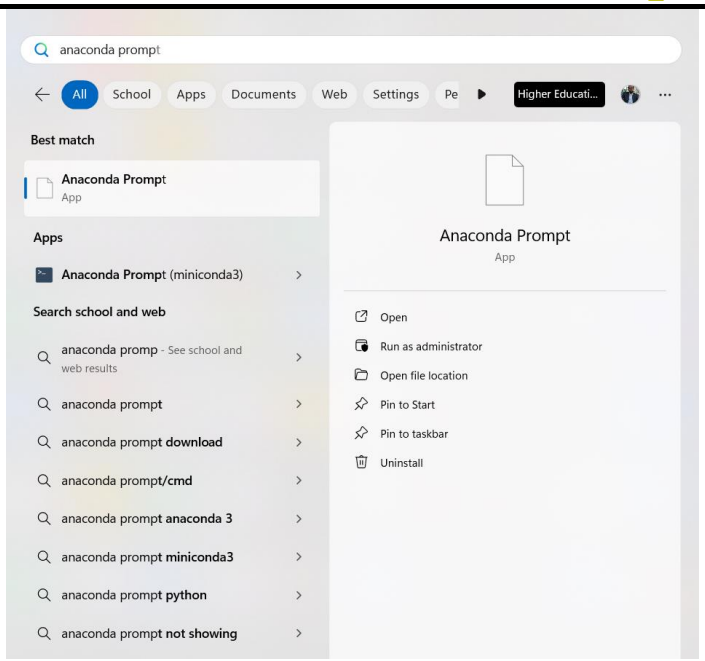


Figure 8 Step 8 for IDE Installation

Configuration

Once Anaconda prompt is opened, run the following commands

```
>> conda create –name=uetai python=3.12

>> conda activate uetai

>> conda install spyder
```

Setting Up and Running the Debugger

1. Launch Spyder from the Anaconda Navigator or by typing spyder in the command line.
2. Create a new file in Spyder and write a simple Python script.

```
def add(a, b):
    result = a + b
    return result

x = 10
y = 20
sum = add(x, y)
print(f"The sum of {x} and {y} is {sum}")
```

3. Click on the left margin next to the line numbers where you want to set a breakpoint. A red dot will appear indicating the breakpoint.
4. Click on the "Debug" button in the toolbar or press Ctrl+F5 to start debugging. Spyder will run the script until it hits the first breakpoint.
5. **Step Through Code**
    - Use the debugging toolbar to step through the code:
        - **Continue**: Resume execution until the next breakpoint (F5).
        - **Step**: Step into the function (F11).
        - **Next**: Step over the function (F10).
        - **Return**: Step out of the function (Shift+F11).
6. **Inspect Variables**
    - Use the "Variable Explorer" to inspect variables at each breakpoint. This window displays the current values of variables in the script.

Basics of Lists, Tuples, and Other Data Structures in Python

Understanding data structures is fundamental for any programming task, including AI and ML. This section covers the basics of lists, tuples, dictionaries, and sets, along with some essential functions and operations.

Lists

Lists are ordered, mutable collections of items. They are defined using square brackets `[]`.

Creating a List:

```
my_list = [1, 2, 3, 4, 5]
```

Accessing Elements:

```
print(my_list[0])  # Output: 1
print(my_list[-1])  # Output: 5 (last element)
```

Modifying Elements:

```
my_list[0] = 10
print(my_list)  # Output: [10, 2, 3, 4, 5]
```

List Functions and Methods:

```
# Adding elements
my_list.append(6)
print(my_list)  # Output: [10, 2, 3, 4, 5, 6]

# Removing elements
my_list.remove(2)
print(my_list)  # Output: [10, 3, 4, 5, 6]

# Inserting elements
my_list.insert(1, 20)
print(my_list)  # Output: [10, 20, 3, 4, 5, 6]

# Length of list
print(len(my_list))  # Output: 6

# Slicing
print(my_list[1:4])  # Output: [20, 3, 4]
```

Tuples

Tuples are ordered, immutable collections of items. They are defined using parentheses `()`.

Creating a Tuple:

```
my_tuple = (1, 2, 3, 4, 5)
```

Accessing Elements:

```
print(my_tuple[0])  # Output: 1
print(my_tuple[-1])  # Output: 5 (last element)
```

Tuple Functions:

```
# Length of tuple
print(len(my_tuple))  # Output: 5
```

```
# Concatenation
new_tuple = my_tuple + (6, 7)
print(new_tuple)  # Output: (1, 2, 3, 4, 5, 6, 7)

# Slicing
print(my_tuple[1:4])  # Output: (2, 3, 4)
```

Dictionaries

Dictionaries are unordered collections of key-value pairs. They are defined using curly braces `{}`.

Creating a Dictionary:

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
```

Accessing Elements:

```
print(my_dict['name'])  # Output: John
print(my_dict.get('age'))  # Output: 25
```

Modifying Elements:

```
my_dict['age'] = 26
print(my_dict)  # Output: {'name': 'John', 'age': 26, 'city': 'New York'}
```

Dictionary Functions and Methods:

```
# Adding elements
my_dict['email'] = 'john@example.com'
print(my_dict)  # Output: {'name': 'John', 'age': 26, 'city': 'New York', 'email':
'john@example.com'}

# Removing elements
my_dict.pop('city')
print(my_dict)  # Output: {'name': 'John', 'age': 26, 'email': 'john@example.com'}

# Length of dictionary
print(len(my_dict))  # Output: 3

# Keys and Values
print(my_dict.keys())  # Output: dict_keys(['name', 'age', 'email'])
print(my_dict.values())  # Output: dict_values(['John', 26, 'john@example.com'])
```

Sets

Sets are unordered collections of unique items. They are defined using curly braces `{}` or the `set()` function.

Creating a Set:

```
my_set = {1, 2, 3, 4, 5}
```

Accessing Elements: Sets do not support indexing or slicing because they are unordered.

Set Functions and Methods:

```
# Adding elements
my_set.add(6)
print(my_set)  # Output: {1, 2, 3, 4, 5, 6}

# Removing elements
my_set.remove(2)
print(my_set)  # Output: {1, 3, 4, 5, 6}

# Length of set
print(len(my_set))  # Output: 5

# Set operations
another_set = {4, 5, 6, 7, 8}
print(my_set.union(another_set))  # Output: {1, 3, 4, 5, 6, 7, 8}
print(my_set.intersection(another_set))  # Output: {4, 5, 6}
print(my_set.difference(another_set))  # Output: {1, 3}
```

Assignment Submission

Your code will be evaluated on two types of test cases, **basic** and **hidden**, which you can see in grader.py/ Basic tests, which are fully provided to you, do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden tests are provided in grader.py, but the correct outputs are not. To run the tests, you will need to have graderUtil.py in the same directory as your code and grader.py. Then, you can run all the tests by typing

python grader.py

This will tell you only whether you passed the basic tests. On the hidden tests, the script will alert you if your code takes too long or crashes, but does not say whether you got the correct output. You can also run a single test (e.g., 1a-0-basic) by typing

python grader.py 1a-0-basic

We strongly encourage you to read and understand the test cases, create your own test cases, and not just blindly run grader.py.

Problems

Problem1: Warming up with foundation

a) Write the function `cube(n)`, which takes in a number and returns its cube. For example, cube(3) => 27.

b) `Write factorial(n)`, which takes in a non-negative integer *n* and returns *n!*, which is the product of the integers from 1 to *n*. (0! = 1 by definition.)

We suggest that you should write your functions so that they raise nice clean errors instead of dying messily when the input is invalid. For example, it would be nice if factorial rejected negative inputs right away; otherwise, you might loop forever. You can signal an error like this: `raise Exception, "factorial: input must not be negative"`

Error handling doesn't affect your lab grade, but on later problems it might save you some angst when you're trying to track down a bug.

c) Implement findAlphabeticallyLastWord, which takes a sentence and find the alphabetically last word from the sentence

findAlphabeticallyLastWord("What is the last word in this sentence?")=> word

d) `count_pattern(pattern lst)`, which counts the number of times a certain pattern of symbols appears in a list, including overlaps. So

```
count_pattern(('a', 'b'), ('a', 'b', 'c', 'e', 'b', 'a', 'b', 'f')) =>2
count_pattern(('a', 'b', 'a'), ('g', 'a', 'b', 'a', 'b', 'a', 'b', 'a')) =>3
```

Problem2: Complexity

One way to measure the complexity of a mathematical expression is the depth of the expression describing it in Python lists. Write a program that finds the depth of an expression.

For example:

- `depth('x') => 0`
- `depth(('expt', 'x', 2)) => 1`
- `depth(('+', ('expt', 'x', 2), ('expt', 'y', 2))) => 2`
- `depth(('/', ('expt', 'x', 5), ('expt', ('-', ('expt', 'x', 2), 1), ('/', 5, 2)))) => 4`

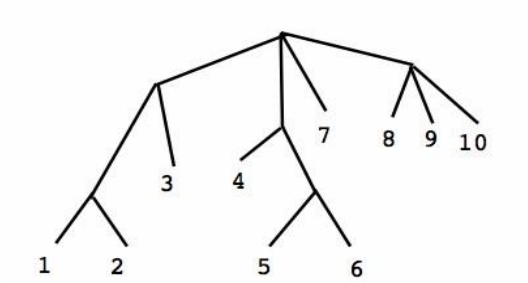
Note that you can use the built-in Python "isinstance()" function to determine whether a variable points to a list of some sort. "isinstance()" takes two arguments: the variable to test, and the type (or list of types) to compare it to. For example:

```
>>> x = [1, 2, 3]
>>> y = "hi!"
>>> isinstance(x, (list, tuple))
True
```

```
>>> isinstance(y, (list, tuple))
False
```

Problem 3: Tree

Your job is to write a procedure that is analogous to list referencing, but for trees. This "tree_ref" procedure will take a tree and an index, and return the part of the tree (a leaf or a subtree) at that index.



For trees, indices will have to be lists of integers. Consider the tree shown, represented by this Python tuple:
`(((1, 2), 3), (4, (5, 6)), 7, (8, 9, 10))`
To select the element 9 out of it, we'd normally need to do something like `tree[3][1]`. Instead, we'd prefer to do `tree_ref(tree, (3, 1))` (note that we're using zero-based indexing, as in list-ref, and that the indices come in top-down order; so an index of (3, 1) means you should take the fourth branch of the main tree, and then the second branch of that subtree). As another example, the element 6 could be selected by `tree_ref(tree, (1, 1, 1))`.

Note that it's okay for the result to be a subtree, rather than a leaf. So `tree_ref(tree, (0,))` should return `((1, 2), 3)`.

Problem 4: Symbolic algebra

Throughout the semester, you will need to understand, manipulate, and extend complex algorithms implemented in Python. You may also want to write more functions than we provide in the skeleton file for a lab.

In this problem, you will complete a simple computer algebra system that reduces nested expressions made of sums and products into a **single sum of products**. For example, it turns the expression `(2 * (x + 1) * (y + 3))` into `((2 * x * y) + (2 * x * 3) + (2 * 1 * y) + (2 * 1 * 3))`. You could choose to simplify further, such as to ((2 * x * y) + (6 * x) + (2 * y) + 6)), but it is not necessary.

This procedure would be one small part of a symbolic math system. You may want to keep in mind the principle of reducing a complex problem to a simpler one.

An algebraic expression can be simplified in this way by repeatedly applying the associative law and the distributive law.

Associative law

((a + b) + c) = (a + (b + c)) = (a + b + c)

((a * b) * c) = (a * (b * c)) = (a * b * c)

Distributive law

((a + b) * (c + d)) = ((a * c) + (a * d) + (b * c) + (b * d))

The code for this part of the lab is in `algebra.py` (available at https://drive.google.com/file/d/1Bs3Yl2t5i0KCExWxMq_M6ff1JSdc7Xwp/view?usp=sharing). It defines an abstract `Expression` class, `Sum` and `Product` expressions, and a method called `Expression.simplify()`. This method starts by applying the associative law for you, but it will fail to perform the distributive law. For that it delegates to a function called `do_multiply` that you need to write. Read the documentation in the code for more details.

This exercise is meant to get you familiar with Python and using it to solve an interesting problem. It is intended to be algorithmically straightforward. You should try to reason out the logic that you need for this function on your own. If you're having trouble expressing that logic in Python, though, don't hesitate to ask3.

Some hints for solving the problem:

- How do you use recursion to make sure that your result is thoroughly simplified?
- In which case should you *not* recursively call `simplify()`?