Unit-2

2.1 Lambda Calculus: Lambda calculus is a formal system in mathematical logic and computer science for expressing computation based on function abstraction and application. It serves as the foundation for functional programming languages like Haskell.

i. **Lambda Abstraction**: A way to define anonymous functions. It's written as:

λx.expression

where '*x'* is the parameter and '*expression'* is the body of the function

For example, λx. x+1

represents a function that takes an argument xxx and returns x+1x + 1x+1.

ii. **Application**: The process of applying a function to an argument. If you have a function λx. x+1 and apply it to 2, it's written as:

(λx. x+1) 2,

This evaluates to 2+1=32 + 1 = 32+1=3.

iii. Variables: Represent placeholders or arguments for functions, like 'x' in λx. x+1.

Syntax

- Variables: x, y, z, etc.
- Abstraction: λx. M, where M is an expression.
- Application: (M N), where M and N are expressions.

2.1.1 Considering the Origins of Lambda Calculus: Lambda calculus was developed by Alonzo Church in the 1930s as a part of his efforts to formalize the concept of computation and address foundational issues in mathematics.

Foundations of Mathematics:

- In the early 20th century, mathematicians were deeply concerned with the foundations of mathematics. The discovery of paradoxes in set theory, such as Russell's paradox, raised questions about the consistency of mathematical systems.
- Mathematicians like David Hilbert proposed formal systems to provide a solid foundation for all of mathematics, aiming to prove that such systems were consistent and complete.

The Quest for Formalism:

- Hilbert's program aimed to formalize all of mathematics by developing a complete and consistent set of axioms. The goal was to show that every mathematical truth could be derived from these axioms using a formal system.
- In parallel, logicians like Kurt Gödel demonstrated the limitations of such formal systems. Gödel's incompleteness theorems showed that in any sufficiently powerful formal system, there are true statements that cannot be proven within the system.

Alonzo Church and Lambda Calculus

Alonzo Church:

- Alonzo Church, an American mathematician and logician, was working on formalizing the concept of functions and computation. He was interested in the notion of effective computability, which is the idea of defining what it means for a function to be computed by a mechanical process.

Development of Lambda Calculus:

- In the early 1930s, Church introduced lambda calculus as a formal system to describe functions and their applications. He used it to provide a foundation for logic and mathematics.
- Lambda calculus was intended to be a simple and universal language for expressing computations. It's based on the idea that functions can be applied to arguments and that this process can be formalized using symbolic manipulation.

Church-Turing Thesis:

- Church's work on lambda calculus, along with Alan Turing's work on Turing machines, led to the formulation of the Church-Turing thesis. This thesis states that any function that can be effectively computed can be computed by a Turing machine or expressed in lambda calculus.

- This thesis essentially equates the notion of "computable function" with functions expressible in lambda calculus or computable by a Turing machine, providing the foundation for modern computer science.

Impact and Legacy

- **Functional Programming**: Lambda calculus influenced the development of functional programming languages, such as Lisp, Haskell, and Scheme. These languages incorporate the concept of first-class functions, higher-order functions, and function composition, all rooted in lambda calculus.
- **Mathematical Logic**: Lambda calculus also influenced the development of type theory and the study of formal systems in mathematical logic.
- **Computational Theory**: The study of lambda calculus contributed to the field of theoretical computer science, particularly in understanding the limits of computation, recursion theory, and the foundations of programming languages.

2.1.2 Understanding the rules: Lambda calculus is governed by a set of rules that define how expressions can be manipulated and evaluated. These rules are essential for understanding how lambda expressions are simplified and how functions are applied.

i. Syntax Rules

- **Variables**: These are symbols that represent values or functions. For example, x, y, and z are variables.
- **Abstraction**: This defines a function. It's written as λx.M, where x is the parameter and M is the body of the function. The variable x is bound in the expression M.
- **Application**: This represents the application of a function to an argument. It's written as (M N), where M is the function and N is the argument.

ii. Reduction Rules describe how expressions in lambda calculus are simplified or evaluated.

α-conversion (Alpha Conversion)

- Purpose: Renames bound variables to avoid conflicts or clashes with other variables.
- Rule**:** You can rename the bound variable in an abstraction, as long as the new name doesn't conflict with any free variables in the expression.

Example: λx. x+1can be renamed to λy. y+1

This is useful for avoiding name clashes during substitutions.

β-reduction (Beta Reduction)

- Purpose**:** The primary rule for function application in lambda calculus, which allows you to apply a function to an argument.
- Rule**:** To apply a function (λx.M) N, replace all instances of the bound variable x in M with N.

Example**:** (λx.x+1) 2 → 2+1

This evaluates to 3.

Note: β-reduction is analogous to function application in programming.

η-conversion (Eta Conversion)

- Purpose: Captures the notion of extensionality in functions. It expresses that two functions are equivalent if they give the same result for all inputs.
- Rule: A function λx.(M x) is equivalent to M if x does not appear free in M.

Example**:** λx.(fx)is equivalent to f

This rule helps in optimizing or simplifying expressions.

iii. Free and Bound Variables Understanding free and bound variables is crucial in lambda calculus.

- Bound Variable: A variable is bound if it is a parameter of a lambda abstraction. For example, in λx.x, x is bound.
- Free Variable: A variable is free if it is not bound by any lambda abstraction in the expression. For example, in λx.y, y is free.

iv. Normal Form

- An expression is in normal form if no further β-reduction is possible. This means the expression cannot be simplified any further.

- Not all lambda expressions have a normal form; some can lead to infinite reduction sequences (e.g., (λx. x x) (λx. x x)).

v. Combinators

- A combinator is a lambda expression with no free variables. It's a closed expression where every variable is bound.

Example: λx. λy. x

This is a combinator because all variables are bound.

vi. Substitution

- Substitution is the process of replacing a variable with an expression. When applying a function (λx.M) N, you substitute x with N in M.

Notation: M[x:=N]

This means that in M, every occurrence of x is replaced with N.

Example: (λx. x+y) 2 → (2+y)

2.1.3 Performing reduction operations: Performing reduction operations in lambda calculus involves simplifying or evaluating lambda expressions according to specific rules like β-reduction, α-conversion, and η-conversion.

i. β-Reduction (Beta Reduction) This is the core reduction rule in lambda calculus. It involves applying a function to an argument.

Example **: Simple β-reduction**

Consider the expression:

(λx. x+1) 2

Step 1**:** Identify the function and the argument.

The function is λx. x+1 The argument is 222.

Step 2**:** Substitute the argument for the bound variable in the function's body. Replace x with 2 in the expression x + 1.

Result: 2+1=3

Example : **Nested β-reduction**

Consider the expression:

(λx. (λy. x+y)) 34

Step 1**:** Apply the outermost function to its argument. The function is λx. (λy. x+y. The argument is 3.

Substitute x with 3 in λy. x+y

Result: (λy.3+y)4

Step 2: Now, apply the resulting function λy. 3+y to '4'

Substitute y with 4.

ii. α-Conversion (Alpha Conversion) This rule allows you to rename the bound variables to avoid clashes.

Example : **α-Conversion**

Consider the expression:

λx. x+1

If we want to avoid using x because it's already used in another part of the expression, we can rename it:

Result: λy.y+1

Note**:** α-conversion is often used to avoid conflicts during substitution in β-reduction.

iii. η-Conversion (Eta Conversion) This rule simplifies functions by removing redundant abstractions.

Example **: η-Conversion** Consider the expression:

λx. (f x) If x does not appear freely in f, this can be simplified using η-conversion.

Result: f

iv. Combining Reductions Let's walk through a more complex example that combines these operations.

Example: Consider the expression:

(λf.(λx.f(xx)) (λx.f(xx))) (λy.y+1)

Step 1: Apply the outermost function to its argument.

The function is λf. (λx. f (x x)) (λx. f (x x))

The argument is λy. y+1

Substitute f with λy. y+1

Result: (λx.(λy.y+1)(xx)) (λx.(λy.y+1)(xx))

Step 2**:** Now, apply the inner function λx. (λy. y+1) (x x).

Substitute x with λx. (λy. y+1) (x x)

2.1.4 Creating lambda functions in haskell and python: Lambda functions are anonymous functions defined without a name.

Haskell lambda functions are defined using the \ symbol (backslash), followed by the parameters, ->, and the function body.

Example1: simple addition

```
add = \x y -> x + y
```

You can then use this lambda function like a regular function:

```
result = add 5 3  -- result will be 8
```

Example2 : Filtering a List You can use a lambda function in a higher-order function like 'filter':

```
filtered = filter (\x -> x > 5) [1, 3, 5, 7, 9]  -- filtered will be [7, 9]
```

Python lambda functions are defined using the lambda keyword, followed by the parameters, a colon ':', and the expression.

Example 1: Simple Addition

```
add = lambda x, y: x + y
```

You can use this lambda function like this:

```
result = add(5, 3)  # result will be 8
```

Example 2: Filtering a List You can also use lambda functions with higher-order functions like 'filter'':

```
filtered = list(filter(lambda x: x > 5, [1, 3, 5, 7, 9]))  # filtered will be [7, 9]
```

- Haskell: Lambda functions are written as \x y -> expression.
- Python: Lambda functions are written as lambda x, y: expression.

2.2 Data Types: Data types are a fundamental concept in programming languages, defining the kind of data that can be stored and manipulated within a program.

2.2.1 Defining List Uses: Lists are fundamental data structures used to store collections of elements. They are versatile and allow various operations like adding, removing, and accessing elements.

Haskell lists are homogeneous, meaning all elements in the list must be of the same type. Lists are defined using square brackets, with elements separated by commas.

Defining a List:

```
-- A list of integers
numbers :: [Int]
numbers = [1, 2, 3, 4, 5]
-- A list of characters (which is also a String)
chars :: [Char]
chars = ['a', 'b', 'c']
-- A list of strings
wordsList :: [String]
wordsList = ["Hello", "World", "Haskell"]
```

List Operations:

- ***Concatenation****:* Combine two lists using the '++' operator.

```
combinedList = [1, 2, 3] ++ [4, 5, 6]  -- Result: [1, 2, 3, 4, 5, 6]
```

- ***Prepend (Cons)****:* Add an element to the front of a list using the ':' operator.

```
newList = 0 : numbers  -- Result: [0, 1, 2, 3, 4, 5]
```

- ***Accessing Elements***: Use the '!!' operator to access an element by index.

```
firstElement = numbers !! 0  -- Result: 1
```

- ***Pattern Matching***: Extract elements from a list.

```
headOfList :: [a] -> a
headOfList (x:_) = x
tailOfList :: [a] -> [a]
tailOfList (_:xs) = xs
```

- ***List Comprehension***: Create lists based on existing lists.

evenNumbers = [x * 2 | x <- [1..10]] -- Result: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

Python lists are heterogeneous, meaning they can contain elements of different types. Lists are defined using square brackets, with elements separated by commas.

Defining a List:

```
# A list of integers
numbers = [1, 2, 3, 4, 5]
# A list of characters (typically strings of length 1)
chars = ['a', 'b', 'c']
# A list of strings
words_list = ["Hello", "World", "Python"]
# A list of mixed types
mixed_list = [1, "two", 3.0]
```

List Operations:

- ***Concatenation****:* Combine two lists using the '+' operator.

```
combined_list = [1, 2, 3] + [4, 5, 6]  # Result: [1, 2, 3, 4, 5, 6]
```

- ***Append****:* Add an element to the end of a list using the 'append()' method.

```
numbers.append(6)  # Result: [1, 2, 3, 4, 5, 6]
```

- ***Accessing Elements***: Use square brackets with an index to access an element.

```
first_element = numbers[0]  # Result: 1
```

- ***Slicing****:* Access a sub-list using the slicing notation.

```
sub_list = numbers[1:4]  # Result: [2, 3, 4]
```

- ***List Comprehension***: Create lists based on existing lists.

```
even_numbers = [x * 2 for x in range(1, 11)]  # Result: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

2.2.2 List Operations: List operations are essential for manipulating and working with lists in both Haskell and Python.

Haskell:

i. List Creation:

- *Literal List:* numbers = [1, 2, 3, 4, 5]
- *Range List*: rangeList = [1..10] -- Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

ii. Accessing Elements:

- *Head:* Get the first element of the list.

```
firstElement = head [1, 2, 3]  -- Result: 1
```

- *Tail:* Get the list without the first element.

```
restOfList = tail [1, 2, 3]  -- Result: [2, 3]
```

- *Indexing***:** Access element by index (0-based).

```
secondElement = [1, 2, 3] !! 1  -- Result: 2
```

- *Last:* Get the last element of the list.

```
lastElement = last [1, 2, 3]  -- Result: 3
```

- *Init***:** Get the list without the last element.

```
initList = init [1, 2, 3]  -- Result: [1, 2]
```

iii. List Operations:

- *Concatenation:* Combine two lists.

```
combinedList = [1, 2, 3] ++ [4, 5, 6]  -- Result: [1, 2, 3, 4, 5, 6]
```

- *Prepend (Cons):* Add an element to the front of a list.

```
newList = 0 : [1, 2, 3]  -- Result: [0, 1, 2, 3]
```

- *Length:* Get the number of elements in the list.
  listLength = length [1, 2, 3] -- Result: 3
- *Reverse*: Reverse the elements in the list.
  reversedList = reverse [1, 2, 3] -- Result: [3, 2, 1]

iv. Higher-Order Functions:

- *Map:* Apply a function to each element of the list.
  *squaredList = map (\x -> x * x) [1, 2, 3] -- Result: [1, 4, 9]*
- *Filter*: Select elements that satisfy a predicate.
  evenNumbers = filter even [1, 2, 3, 4, 5] -- Result: [2, 4]
- *Fold (Reduce):* Reduce a list to a single value using a binary function.
  sumOfList = foldl (+) 0 [1, 2, 3] -- Result: 6

v. List Comprehensions:

- *Generate Lists:*
  *squares = [x * x | x <- [1..5]] -- Result: [1, 4, 9, 16, 25]*
- *Filter with List Comprehension:*
  *evenSquares = [x * x | x <- [1..5], even x] -- Result: [4, 16]*

Python:

i. List Creation:

- *Literal List:* numbers = [1, 2, 3, 4, 5]
- Range List (using `range( )`): range_list = list(range(1, 11)) Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

ii. Accessing Elements:

- *Indexing*: Access element by index (0-based). second_element = [1, 2, 3] [1] Result: 2
- *Slicing*: Access a sublist. sub_list = [1, 2, 3, 4, 5] [1:4] Result: [2, 3, 4]
- *Negative Indexing*: Access elements from the end. last_element = [1, 2, 3] [-1] Result: 3

iii. List Operations:

- *Concatenation*: Combine two lists. combined_list = [1, 2, 3] + [4, 5, 6] Result: [1, 2, 3, 4, 5, 6]
- *Append*: Add an element to the end of a list. numbers.append(6) Result: [1, 2, 3, 4, 5, 6]
- *Insert*: Add an element at a specific index. numbers.insert(2, 'a') Result: [1, 2, 'a', 3, 4, 5, 6]
- *Length*: Get the number of elements in the list. list_length = len([1, 2, 3]) Result: 3
- *Remove*: Remove the first occurrence of an element. numbers.remove(3) Result: [1, 2, 'a', 4, 5, 6]
- *Pop:* Remove and return an element at a specific index (default is the last element).
  last_element = numbers.pop() # Removes 6, Result: [1, 2, 'a', 4, 5]
- *Reverse*: Reverse the elements in the list. numbers.reverse() Result: [5, 4, 'a', 2, 1]

iv. Higher-Order Functions:

- *Map*: Apply a function to each element of the list.
  squared_list = list(map(lambda x: x * x, [1, 2, 3])) # Result: [1, 4, 9]
- *Filter:* Select elements that satisfy a predicate
  even_numbers = list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5])) # Result: [2, 4]
- *Reduce* (from `functools` module): Reduce a list to a single value using a binary function.
  from functools import reduce
  sum_of_list = reduce(lambda x, y: x + y, [1, 2, 3]) Result: 6

v. List Comprehensions:

- *Generate Lists*: squares = [x * x for x in range(1, 6)] Result: [1, 4, 9, 16, 25]
- *Filter with List Comprehension*: even_squares = [x * x for x in range(1, 6) if x % 2 == 0] Result: [4, 16]

2.2.3 Dictionary: Dictionaries (also known as associative arrays or hash maps) are data structures that store key-value pairs, allowing efficient retrieval, insertion, and deletion of elements based on their keys. Dictionaries are a fundamental part of many programming languages, including Python, while Haskell approaches similar problems using other constructs like Data.Map.

Python: Dictionaries

In Python, dictionaries are a built-in data type, defined using curly braces `{}` with keys and values separated by colons `:`.

i. Defining a Dictionary:

```
# Empty dictionary
empty_dict = {}
```

```
# Dictionary with key-value pairs
person = {
  "name": "Alice",
 "age": 30,
 "city": "New York"
}
```

ii. Accessing Elements:

- Get Value by Key: Access values using their keys. *name = person["name"] Result: "Alice"*
- Using **`get()`**: A safer way to access values, providing a default if the key is not found.

```
age = person.get("age", 25)    Result: 30
occupation = person.get("occupation", "Unknown")    Result: "Unknown"
```

iii. Modifying a Dictionary:

- Adding or Updating Elements: Add a new key-value pair or update an existing key's value

```
person["age"] = 31   Updates the value of "age" to 31
person["occupation"] = "Engineer"   Adds a new key-value pair
```

- Removing Elements: Use del or pop() to remove elements by key.

```
del person["city"]  # Removes the "city" key-value pair
occupation = person.pop("occupation")  # Removes and returns the value of "occupation"
```

iv. Dictionary Operations:

- Check Existence**:** Use the `in` keyword to check if a key exists *has_age = "age" in person Result: True*
- *Iterating Over Keys and Values*

```
for key in person:
    print(key, person[key])
for key, value in person.items():
    print(f"{key}: {value}")
```

- *Get All Keys or Values*

```
keys = person.keys()        Returns a list of all keys
values = person.values()    Returns a list of all values
```

- Dictionary Comprehensions**:** Create a new dictionary based on existing data

*squared_numbers = {x: x*x for x in range(1, 6)} Result: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}*

v. Merging Dictionaries:

- **Using `update()`**: Merge another dictionary into the current dictionary

```
additional_info = {"hobby": "reading", "city": "San Francisco"}
person.update(additional_info)
```

- Using the '**|**'Operator (Python 3.9+): *person = person | {"country": "USA"}*

Haskell: Data.Map

Haskell doesn't have a built-in dictionary type like Python, but it provides a similar functionality through the `Data.Map` module. This module offers efficient map operations, where keys are unique and associated with values.

i. Importing 'Data.Map': *import qualified Data.Map as Map*

ii.Creating a Map:

- From List of Pairs**:**

```
person :: Map.Map String String
person = Map.fromList [("name", "Alice"), ("age", "30"), ("city", "New York")]
```

- *Empty Map:* *emptyMap = Map.empty*

iii. Accessing Elements:

- Lookup by Key**:** Returns '`Maybe`' type ('`Just value` or `Nothing`')

name = Map.lookup "name" person -- Result: Just "Alice"

iv. Modifying a Map:

- Insert or Update Elements: *updatedPerson = Map.insert "age" "31" person*
- *Deleting Elements:* *personWithoutCity = Map.delete "city" person*

v. Map Operations:

- Check Existence**:** *hasAge = Map.member "age" person -- Result: True*
- *Get All Keys or Values:*

```
keys = Map.keys person  -- Result: ["age", "city", "name"]
values = Map.elems person  -- Result: ["30", "New York", "Alice"]
```

- ***Merge Maps***

person2 = Map.fromList [("hobby", "reading"), ("city", "San Francisco")]
mergedPerson = Map.union person person2

2.2.4 Set: Sets are collections of unique elements, often used to perform mathematical set operations like union, intersection, and difference. They are available as a built-in data type in Python, while in Haskell, sets are typically handled using the Data.Set module.

Python: Sets In Python, sets are unordered collections of unique elements. They are defined using curly braces {} or the set() function.

i. Defining a Set

- *Using Curly Braces: fruits = {"apple", "banana", "cherry"}*
- *Using the `set( )` Constructor:*

numbers = set([1, 2, 3, 4, 5])
empty_set = set() # Note: {} creates an empty dictionary, not a set

ii. Adding and Removing Elements:

- *Add an Element*: *fruits.add("orange") Result: {"apple", "banana", "cherry", "orange"}*
- *Remove an Element:*

fruits.remove("banana") # Removes "banana", raises KeyError if not found
fruits.discard("banana") # Safely removes "banana", does nothing if not found

- *Pop an Element (removes and returns an arbitrary element):*

random_fruit = fruits.pop() Result: Removes a random element from the set

iii. Set Operations:

- *Union*: Combine elements from both sets (all unique elements)

set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2 # Result: {1, 2, 3, 4, 5}

- *Intersection*: Elements common to both sets. intersection_set = set1 & set2 Result: {3}
- *Difference*: Elements in the first set but not in the second difference_set = set1 - set2 Result: {1, 2}
- Symmetric Difference: Elements in either set, but not in both

sym_diff_set = set1 ^ set2 # Result: {1, 2, 4, 5}

iv. Other Set Operations:

- *Subset and Superset:*

is_subset = {1, 2} <= set1 Result: True
is_superset = set1 >= {1, 2} Result: True

- *Check for Membership: is_member = 1 in set1 Result: True*
- *Set Comprehensions: squared_set = {x * x for x in range(5)} # Result: {0, 1, 4, 9, 16}*

Haskell: Data.Set Haskell doesn't have a built-in set type, but it provides similar functionality through the Data.Set module, which is part of the standard library. Data.Set handles collections of unique elements efficiently.

i. Importing `Data.Set`: import qualified Data.Set as Set

ii. Creating a Set:

- From List: fruits = Set.fromList ["apple", "banana", "cherry"]
- Empty Set: emptySet = Set.empty

iii. Adding and Removing Elements

- Insert an Element: fruitsWithOrange = Set.insert "orange" fruits
- Delete an Element: fruitsWithoutBanana = Set.delete "banana" fruits

iv. Set Operations:

- Union: Combine elements from both sets (all unique elements).

set1 = Set.fromList [1, 2, 3]
set2 = Set.fromList [3, 4, 5]
unionSet = Set.union set1 set2 -- Result: fromList [1, 2, 3, 4, 5]

- Intersection: Elements common to both sets

intersectionSet = Set.intersection set1 set2 -- Result: fromList [3]

- Difference: Elements in the first set but not in the second

differenceSet = Set.difference set1 set2 -- Result: fromList [1, 2]

- Symmetric Difference: Elements in either set, but not in both

symDiffSet = Set.symmetricDifference set1 set2 -- Result: fromList [1, 2, 4, 5]

***v.* Other Set Operations**

- Subset and Superset:

isSubset = Set.isSubsetOf (Set.fromList [1, 2]) set1 -- Result: True

isSuperset = Set.isSupersetOf set1 (Set.fromList [1, 2]) -- Result: True

- Check for Membership**:** isMember = Set.member 1 set1 -- Result: True
- Convert Set to List listOfFruits = Set.toList fruits -- Result: ["apple", "banana", "cherry"]

2.2.5 String operations: are fundamental in programming for manipulating and processing text. Both Python and Haskell offer rich sets of functions and methods to perform various string operations.

Python: String Operations are sequences of characters and are immutable, meaning they cannot be changed after they are created. Python provides numerous methods to perform operations on strings.

i. Creating Strings:

- Single and Double Quotes:

single_quoted = 'Hello'

double_quoted = "World"

- Multi-line Strings:

multi_line = """This is

a multi-line

string."""

ii. Accessing Characters:

- Indexing (0-based): char = "Hello"[1] Result: 'e'
- Slicing: sub_string = "Hello"[1:4] Result: 'ell'
- Negative Indexing: last_char = "Hello"[-1] Result: 'o'

***iii.* String Concatenation:**

- Using '+': full_string = "Hello" + " " + "World" Result: 'Hello World'
- Using `join()` (for multiple strings):

words = ["Hello", "World"]

sentence = " ".join(words) Result: 'Hello World'

***iv.* String Length:**

- Using `len()` function: length = len("Hello") Result: 5

v. String Case Conversion:

- Uppercase and Lowercase:

upper = "hello".upper() Result: 'HELLO'

lower = "HELLO".lower() Result: 'hello'

- Title Case: title = "hello world".title() Result: 'Hello World'
- Capitalize First Letter: capitalized = "hello".capitalize() Result: 'Hello'

vi. Trimming and Stripping:

- Remove Leading and Trailing Whitespace: stripped = " hello ".strip() # Result: 'hello'
- Remove Leading or Trailing Characters

stripped_leading = " hello ".lstrip() Result: 'hello '

stripped_trailing = " hello ".rstrip() Result: ' hello'

vii. Searching and Replacing:

- Finding Substrings index = "Hello World".find("World") Result: 6
- Checking Presence is_present = "World" in "Hello World" Result: True
- Replacing Substrings replaced = "Hello World".replace("World", "Python") Result: 'Hello Python'

viii. Splitting and Joining

- Splitting a String into a List: words = "Hello World".split() Result: ['Hello', 'World']
- Joining a List into a String: sentence = " ".join(["Hello", "World"]) Result: 'Hello World'

Haskell: String Operations are typically represented as lists of characters ([Char]), which means you can use list operations on them. The Data.Text or Data.String modules also provide more efficient string operations.

i. Defining Strings:

- Single-line Strings**:** *greeting = "Hello, World!"*
- Multi-line Strings**:** *multiLine = "This is a\nmulti-line string."*

ii. Accessing and Slicing Strings:

- Indexing: Access characters by their index firstChar = greeting !! 0 -- Result: 'H'
- Slicing: Extract a substring (requires custom function)

takeHello = take 5 greeting -- Result: "Hello"
dropHello = drop 7 greeting -- Result: "World!"

iii. String Concatenation:

- *Using '++' Operator fullGreeting = "Hello" ++ ", " ++ "World!"*

iv. Repeating Strings:

- Using concat and 'replicate' repeated = concat (replicate 3 "Ha") -- Result: "HaHaHa"

v. String Functions:

- Changing Case (requires Data.Char)

import Data.Char (toUpper, toLower)
upperCase = map toUpper greeting -- Result: "HELLO, WORLD!"
lowerCase = map toLower greeting -- Result: "hello, world!"

- Trimming Whitespace (requires Data.Text)

import qualified Data.Text as T
stripped = T.strip " Hello " -- Result: "Hello"

- Replacing Substrings (requires Data.Text)

replaced = T.replace "World" "there" (T.pack greeting) -- Result: "Hello, there!"

- Splitting and Joining Strings (requires Data.List)

import Data.List (intercalate)
splitWords = words greeting -- Result: ["Hello,", "World!"]
joinedWords = intercalate " " splitWords -- Result: "Hello, World!"

- Finding Substrings

import Data.List (isInfixOf)
hasWorld = "World" `isInfixOf` greeting -- Result: True

vi. String Formatting:

- Using `printf` (requires Text.Printf)

import Text.Printf
formatted = printf "Hello, %s!" "Alice" -- Result: "Hello, Alice!"

vi. **Checking String Properties:**

- Check if String is Numeric, Alphabetic (requires Data.Char)

import Data.Char (isDigit, isAlpha)
allDigits = all isDigit "12345" -- Result: True
allAlpha = all isAlpha "Hello" -- Result: True

2.3 Performing pattern matching: Pattern matching is a powerful feature used in many programming languages, allowing developers to destructure data and execute code based on specific patterns. This feature is particularly strong in functional programming languages like Haskell, but it also appears in a limited form in Python.

2.3.1 Looking for patterns in data: the goal is to identify recurring structures, trends, or anomalies that can provide insights or inform decisions. Both Python and Haskell offer tools and techniques for pattern discovery, though Python's libraries are more geared towards data analysis and machine learning, while Haskell's functional programming approach is useful for defining and recognizing patterns programmatically.

Python: Data Pattern Discovery Python is widely used for data analysis, and its libraries are designed to facilitate the discovery of patterns in data.

i. Using Pandas for Data Exploration: Pandas is a powerful library for data manipulation and analysis.

- Loading Data:

```
import pandas as pd
df = pd.read_csv('data.csv')  # Load data from a CSV file
```

- Exploring Data:

```
print(df.head())  # Display the first few rows of the dataset
print(df.describe())  # Summary statistics
print(df.info())  # Information about data types and missing values
```

- Identifying Correlations:

```
correlations = df.corr()  # Calculate correlation matrix
print(correlations)
```

- Finding Unique Values and Frequency:

```
unique_values = df['column_name'].value_counts()
print(unique_values)  # Count of unique values in a column
```

ii. Using Visualization Libraries (Matplotlib, Seaborn): Visualizations help in identifying patterns visually.

- Plotting Distributions:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.histplot(df['column_name'], kde=True)  # Histogram with KDE
plt.show()
```

- Scatter Plots for Relationships:

```
sns.scatterplot(x='feature1', y='feature2', data=df)
plt.show()
```

- Heatmaps for Correlation

```
sns.heatmap(correlations, annot=True, cmap='coolwarm')
plt.show()
```

iii. Using Machine Learning for Pattern Recognition Machine learning models can be used to discover complex patterns in data.

- Clustering (e.g., K-Means):

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3)
df['Cluster'] = kmeans.fit_predict(df[['feature1', 'feature2']])
sns.scatterplot(x='feature1', y='feature2', hue='Cluster', data=df)
plt.show()
```

- Anomaly Detection:

```
from sklearn.ensemble import IsolationForest
model = IsolationForest(contamination=0.1)
df['Anomaly'] = model.fit_predict(df[['feature1', 'feature2']])
sns.scatterplot(x='feature1', y='feature2', hue='Anomaly', data=df)
plt.show()
```

iv. Time Series Analysis For sequential data, time series analysis helps in identifying trends, seasonal patterns, and anomalies.

- Decomposing Time Series:

```
from statsmodels.tsa.seasonal import seasonal_decompose
result = seasonal_decompose(df['time_series_column'], model='additive')
result.plot()
plt.show()
```

- Autocorrelation and Partial Autocorrelation:

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
plot_acf(df['time_series_column'])
plt.show()
plot_pacf(df['time_series_column'])
plt.show()
```

Pattern Discovery in Data Haskell, while not as commonly used in data science as Python, is strong in symbolic pattern matching, functional transformations, and custom algorithm development.

i. List Processing and Pattern Matching: Haskell's list processing capabilities allow you to identify patterns in sequences of data.

- Finding Sequences or Subpatterns:

```
findPattern :: Eq a => [a] -> [a] -> Bool
findPattern pattern = any (pattern `isPrefixOf`) . tails
-- Example: findPattern "abc" "xabcx" returns True
```

- Defining Custom Pattern Matching Functions:

```
matchPattern :: Eq a => [a] -> [a] -> [Int]
matchPattern pattern list = [i | (i, l) <- zip [0..] (tails list), pattern `isPrefixOf` l]
-- Example: matchPattern "ab" "abcab" returns [0, 3]
```

ii. Functional Programming for Pattern Transformation: Haskell's functions can be composed to transform data and identify patterns.

- Mapping and Folding:

```
countOccurrences :: Eq a => a -> [a] -> Int
countOccurrences x = length . filter (== x)
-- Example: countOccurrences 'a' "banana" returns 3
```

- Building Custom Pattern Recognition Algorithms:

```
-- Custom function to detect patterns in a list
  detectPattern :: (a -> Bool) -> [a] -> [a]
  detectPattern predicate = filter predicate
-- Example: detectPattern (> 3) [1, 2, 3, 4, 5] returns [4, 5]
```

iii. Using Libraries for Data Analysis: Although not as extensive as Python's ecosystem, Haskell has libraries like cassava for CSV parsing, HMatrix for numerical computations, and Chart for visualizations.

- Simple CSV Data Processing:

```
import qualified Data.ByteString.Lazy as BL
import qualified Data.Csv as Csv
import Data.Vector (Vector)
processCSV :: FilePath -> IO ()
processCSV filePath = do
csvData <- BL.readFile filePath
case Csv.decode Csv.NoHeader csvData of
Left err -> putStrLn err
Right v -> print (v :: Vector (Vector String))
```

2.3.2 Regular expressions: Regular expressions (regex) are powerful tools for pattern matching within strings. They allow you to search, match, and manipulate text based on specific patterns. Both Python and Haskell support regular expressions, but their usage and libraries differ slightly.

Python: Regular Expressions Python provides the re module for working with regular expressions.

i. Basic Usage of Regular Expressions in Python:

- Importing the **re** Module: import re
- Searching for a Pattern:

```
text = "The quick brown fox jumps over the lazy dog."
match = re.search(r'\bfox\b', text)
if match:
print("Found:", match.group())                Output: Found: fox
```

- Finding All Matches:

```
matches = re.findall(r'\b\w{3}\b', text)  # Finds all 3-letter words
print(matches)  # Output: ['The', 'fox', 'the', 'dog']
```

- Replacing Patterns:

```
new_text = re.sub(r'lazy', 'energetic', text)
print(new_text)  # Output: The quick brown fox jumps over the energetic dog.
```

- Splitting a String by a Pattern:

```
split_text = re.split(r'\s+', text)  # Splits by any whitespace
 print(split_text)  # Output: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog.']
```

ii. **Common Regular Expression Patterns:**

- '**\d**': Matches any digit, equivalent to `[0-9]`.
- '**\D**': Matches any non-digit character.
- '**\w**': Matches any word character (alphanumeric + underscore).
- '**\W**': Matches any non-word character.
- '**\s**': Matches any whitespace character (spaces, tabs, line breaks).
- '**\S**': Matches any non-whitespace character.
- '**^**': Matches the start of a string.
- '**$**': Matches the end of a string.
- '**.***': Matches any character (except line breaks) zero or more times.
- '**[abc]**': Matches any one of the characters a, b, or c.
- '**[a-z]**': Matches any lowercase letter.
- '**[A-Z]**': Matches any uppercase letter.
- '**(pattern)**': Groups patterns together.

Examples:

- Matching an Email Address

```
email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
email = "user@example.com"
match = re.search(email_pattern, email)
if match:
print("Valid email:", match.group())          Output: Valid email: user@example.com
```

- Matching a Date Format:

```
date_pattern = r'\b\d{2}/\d{2}/\d{4}\b'
date = "The event is on 12/25/2023."
match = re.search(date_pattern, date)
if match:
print("Found date:", match.group())          Output: Found date: 12/25/2023
```

Haskell: Regular Expressions Haskell provides several libraries for working with regular expressions, such as regex-posix, regex-tdfa, and regex-pcre. Here's how you can use regex in Haskell:

i. ***Basic Usage of Regular Expressions in Haskell:***

- Importing the Regex Library: *import Text.Regex.Posix -- Using the POSIX regex library*
- Searching for a Pattern:

```
let text = "The quick brown fox jumps over the lazy dog."
let match = text =~ "\\bfox\\b" :: Bool
print match  -- Output: True
```

- Finding All Matches:

```
let matches = getAllTextMatches $ text =~ "\\b\\w{3}\\b" :: [String]
print matches  -- Output: ["The", "fox", "the", "dog"]
```

- Replacing Patterns:

```
import Text.Regex.Posix
import Data.List (intercalate)
let newText = subRegex (mkRegex "lazy") text "energetic"
print newText  -- Output: "The quick brown fox jumps over the energetic dog."
```

ii. Using the `regex-tdfa` Library: The regex-tdfa library is more flexible and often preferred for Haskell.

- Installing the Library: You can install it using Cabal or Stack: cabal install regex-tdfa
- Basic Matching:

```
import Text.Regex.TDFA
let text = "The quick brown fox jumps over the lazy dog."
let match = text =~ "\\bfox\\b" :: Bool
print match  -- Output: True
```

- Extracting Matches:

```
let matches = getAllTextMatches (text =~ "\\b\\w{3}\\b" :: AllTextMatches [] String)
print matches  -- Output: ["The", "fox", "the", "dog"]
```

2.3.3 Pattern Matching in Haskell and python: Pattern matching is a powerful feature in both Haskell and Python, though it is more central to Haskell's design and has been recently introduced in Python.

Haskell In Haskell, pattern matching is a fundamental feature, especially when working with algebraic data types (ADTs) like lists, tuples, or custom data types. It's used extensively in function definitions, case expressions, and list comprehensions.

Example 1: *Pattern Matching with Lists:*

```
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList(xs)
```

- `sumList [] = 0`: This pattern matches an empty list and returns 0.
- `sumList (x:xs) = x + sumList(xs)`: This pattern matches a list with a head x and a tail xs, recursively summing the elements.

Example 2: Pattern Matching with Tuples:

```
describeTuple :: (Int, Int) -> String
describeTuple (0, 0) = "Both are zero"
describeTuple (0, _) = "First is zero"
describeTuple (_, 0) = "Second is zero"
describeTuple _ = "Neither are zero"
```

- This function matches different forms of tuples, handling each case with a specific response.

Example 3: Pattern Matching in Case Expressions:

```
describeNumber :: Int -> String
describeNumber n = case n of
0 -> "Zero"
1 -> "One"
 _ -> "Some other number"
```

- Here 'case' is used to match against different values of 'n'.

Python introduced pattern matching in PEP 634, and it's used similarly to Haskell but with some Python-specific syntax and features. The 'match' statement allows you to match against values and structures.

Example 1: Pattern Matching with Lists:

```
def sum_list(lst):
match lst:
case []:
return 0
case [x, *xs]:
return x + sum_list(xs)
print(sum_list([1, 2, 3]))  # Output: 6
```

- 'case []:' matches an empty list.
- 'Case [x, *xs]:' matches a list with head 'x' and tail 'xs'.

Example 2: Pattern Matching with Tuples:

```
def describe_tuple(tpl):
 match tpl:
 case (0, 0):
    return "Both are zero"
 case (0, _):
    return "First is zero"
 case (_, 0):
    return "Second is zero"
 case _:
    return "Neither are zero"
print(describe_tuple((0, 1)))  # Output: First is zero
```

- 'match' is used similarly to the case expression in Haskell, handling tuples.

Example 3: Pattern Matching with Classes (Python-Specific):

```
class Point:
 def __init__(self, x, y):
    self.x = x
    self.y = y
 def describe_point(point):
 match point:
 case Point(0, 0):
      return "Origin"
 case Point(x, y) if x == y:
      return f"On the line y=x"
 case Point(x, y):
      return f"Point at ({x}, {y})"
  p = Point(1, 1)
print(describe_point(p))  # Output: On the line y=x
```

- Python's pattern matching can match class instances, and it can use guards (if conditions) to further refine matches.