Discrete Structures: Data Containers CMPSC 102
Oliver BONHAM-CARTER
Exam Ahead
Let's Discuss
Definition
Benefits
Summations
Products
Play Time
Higher-Order Sequence Functions
Map
Filter
Reduce
Conclusions
Ask yourself

Discrete Structures: Data Containers CMPSC 102

Oliver BONHAM-CARTER

Fall 2022
Week 8
Slides 02

What to study

- Date: 26 Oct 2022, After Gator Day
- During class, open notes
- **Study: Slides, notes, with chapters to add detail to class material**
- Python basics and code
  - Determining output
  - Picking out bugs from code; fixing code
  - Study the code from the practicals and material covered in class to understand the how programs worked.
  - Lambda functions, lists, dictionaries, n-tuples
  - `for` and `while` loops
  - Iterations over sequences
  - Sequences, strings, sets
  - Conditional statements
  - And other concepts covered during class

Let's Discuss

Key Questions

How do I employ the mathematical concepts of **sequences**, **monoids**, and **lists** to implement efficient Python programs that use functions with a **clearly specified behavior** to perform tasks like finding a name in a file or computing the arithmetic mean of data values?

Learning Objectives

To **remember** and **understand** some the concept of a **monoid**, seeing how it connects to **practical applications** with strings and sequences

A Quick Definition

Monoid Definition

In Abstract Algebra, a **monoid** is a set equipped with an **associative** binary operation and an **identity** element. For example, the non-negative integers with addition form a monoid, the identity element being 0.

- A monoid is a combination of an object (a,b,c) and an operation (+) that meets the following conditions
  - the operation on two of the objects produces a new object of the same kind
    - int + int = int
  - associative operations
    - (a+b) + c = a + (b+c)
  - a null object e must exist, such that $e + a = a + e = a$
    - $n + 0 = n$

What are the benefits of the monoid concept?

- Generalizes the behavior of structures
- Offers an archetype for understanding
- Logical foundation for approach to code
- And provides a better and more logical flow to your code for others to follow?!

Summations
Adding

```
standard_list = [1, 2, 3, 4, 5]
reversed_list = [5, 4, 3, 2, 1]

sum_list = sum(standard_list)
sum_reversed_list = sum(reversed_list)
```

- Summation(i.e, **adding**): Remember that the order does not matter for positive values being added
- `sum` is a built-in function provided by Python and is used for lists
- What is the output of this program segment?

Products
Multiplying

```
import math
standard_list = [1, 2, 3, 4, 5]
reversed_list = [5, 4, 3, 2, 1]
product_list = math.prod(standard_list)
product_reversed_list = math.prod(reversed_list)
```

- Products (i.e, **multiplying**): Remember that the order does not matter for positive values being added
- `math.prod` is a built-in function provided by Python's math library and is used for lists
- What is the output of this program segment?

Application
CVS data

CSV File Containing Population Data

```
1972-01-01,84.700
1973-01-01,85.500
1974-01-01,86.100
1975-01-01,87.000
1976-01-01,87.600
1977-01-01,87.600
1978-01-01,88.000
```

- CSV file stores ordered pairs of **dates** and **population counts**
- Both lists and tuples are examples of **sequences**
- A tuple is an **immutable** data container
- A list is a **mutable** data container
- What are the **trade-offs** when using these containers?

Using Mutable Lists in Python

```
data_number_list = []
  for line in data_text.splitlines():
      ordered_pair = line.split(",")
      data_number_list.append(float(
              ordered_pair[1]))
  return data_number_list
```

- This source code parses the CSV file and extracts content
- What is the purpose of `ordered_pair[1]` ?
- Does this source code use a `tuple` or a `list`?
- What are the differences between `lists` and `tuples`?

Playing With Code

Data from the "file"

```
data_text = """1972-01-01,84.700
1973-01-01,85.500
1974-01-01,86.100
1975-01-01,87.000
1976-01-01,87.600
1977-01-01,87.600
1978-01-01,88.000
"""
print(data_text)
data_text
```

- What does this code do?

Playing With Code

```
for line in data_text.splitlines():
    print(f"line:: {type(line)}") #str
```

- What does this code do?

Separate the string

```
data_number_list = []
for line in data_text.splitlines():
      ordered_pair = line.split(",")
      print(f"ordered_pair = {ordered_pair}")
```

- What does this code do?

Playing With Code

Make a list of data

```
data_number_list = []
for line in data_text.splitlines():
    ordered_pair = line.split(",")
    data_number_list.append(float(ordered_pair[1]))
print(f"data_number_list == {data_number_list}")
```

- What does this code do?

Playing With Code

import

```
import math
print(f"data_number_list == {data_number_list}")
```

Sum

```
sum(data_number_list)
```

Product

```
math.prod(data_number_list)
```

- What does this code do?

Challenges When Using CSF Files?

What could possibly go wrong?!

Discrete Structures: Data Containers CMPSC 102

Oliver BONHAM-CARTER

Exam Ahead

Let's Discuss

Definition

Benefits

Summations

Products

Play Time

Higher-Order Sequence Functions

Map

Filter

Reduce

Conclusions

Ask yourself

Data from the "file"

```
data_text = """1972-01-01,84.700
1973-01-01,85.500
1974-01-01|86.100
1975-01-01;87.000
1976-01-01,
87.600
87.600;1977-01-01
1978-01-01,88.000
"""
print(data_text)
data_text
```

- Handling missing values or values with delimiters
- Parsing files with corrupted data values
- Difficult to efficiently parse large CSV files

Higher-Order Sequence Functions

- Functions that *work* for **any sequence**?
- These **Higher Order** functions should work for lists, ordered pairs, tuples:
  - `map`: Apply a function to every element of a sequence
  - `filter`: Apply a boolean function to every element of a sequence, returning only those matching the filter's rules
  - `reduce`: Apply a function that acts like a binary operator to a sequence of values, combining them to a single value
- These three operators give a **vocabulary** for implementing complex, yet easy-to-ready programs in a functional programming style
- These functions are **higher-order** because they accept function as input

Map Function with a Literal Tuple

```
def square(value):
    return value * value

def map(callFunction, sequence):
    result = (  )
    for element in sequence:
        result += ( callFunction(element), )
    return result

squared = map(square, (2, 3, 5, 7, 11))
print(squared)
```

- What does this code do?

Map Function with a Range Sequence

```
def square(value):
    return value * value

def map(callFunction, sequence):
    result = (  )
    for element in sequence:
        result += ( callFunction(element), )
    return result

squared_range = map(square, range(10))
print(squared_range)
```

- What does this code do?

Filtering Even Numbers from a Tuple

```
def is_even(value):
    if value % 2 == 0:
        return True
    return False

filtered_even = filter(is_even,
    (2, 3, 4, 5, 7, 11))
print(list(filtered_even))
```

- What does this code do?

Filtering Odd Numbers from a Tuple

One way to do it ...

```
def is_even(value):
    if value % 2 != 0:
        return True
    return False

filtered_even = filter(is_even,
    (2, 3, 4, 5, 7, 11))
print(list(filtered_even))
```

- What does this code do?
- How to modify this code to find **another** way?

Summations By Using Reduce

```
def plus(number_one, number_two):
    return number_one + number_two

def reduce(callFunction, sequence, initial):
    result = initial
    for value in sequence:
        result = callFunction(result, value)
    return result

numbers = [1, 2, 3, 4, 5]
added_numbers = reduce(plus, numbers, 0)
print(f"Added numbers: {added_numbers}")
```

- What does this code do?

Monoids and Map-Filter-Reduce

- **Higher-order sequence functions** are **independent** and free of *side effects* and thus can be **parallelized**
- Since a **monoid** has the associativity property, can use `map`, `filter`, and `reduce` operators in **parallel** and then combine the solution, often achieving a **speedup**. This makes the program more efficient!

Monoids and Map-Filter-Reduce

- These three operators give a **vocabulary** for implementing complex, yet easy-to-read, programs in a **functional** programming style
- Map-Filter-Reduce enables **parallel** computation, which is important given the **diminishing** returns associated with sequential computation
- If you can prove that a structure and operation is a **monoid** then you can use **map**, **reduce**, and **filter** to **parallelize** its computations

Monoids and Map-Filter-Reduce

- Monoids are frequently used in Python programs
- Python programs can use higher-order sequence functions
- Using **monoids** and **higher-order** sequence functions:
  1. What is the difference between a list and a tuple?
  2. How does a monoid generalize strings and integers?
  3. How do higher-order sequence functions use monoids?
  4. How can map-filter-reduce support parallel programming?
  5. What type of speedup will a parallel program achieve?
- What are the ways in which the mathematical concept of a monoid connects to a wide variety of **practical applications** in the area of **parallel computing**?
- How does the concept of a **monoid** create an **archetype** in our minds?