Discrete Structures: Monoids CMPSC 102

Oliver BONHAM-CARTER

Spring 2024
Week 6
Slides 01

Please create your gradebook respository!!

`https://classroom.github.com/a/a0KQdhxE`

What to study

- Date: 18^{th} March 2024, During Lab, 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

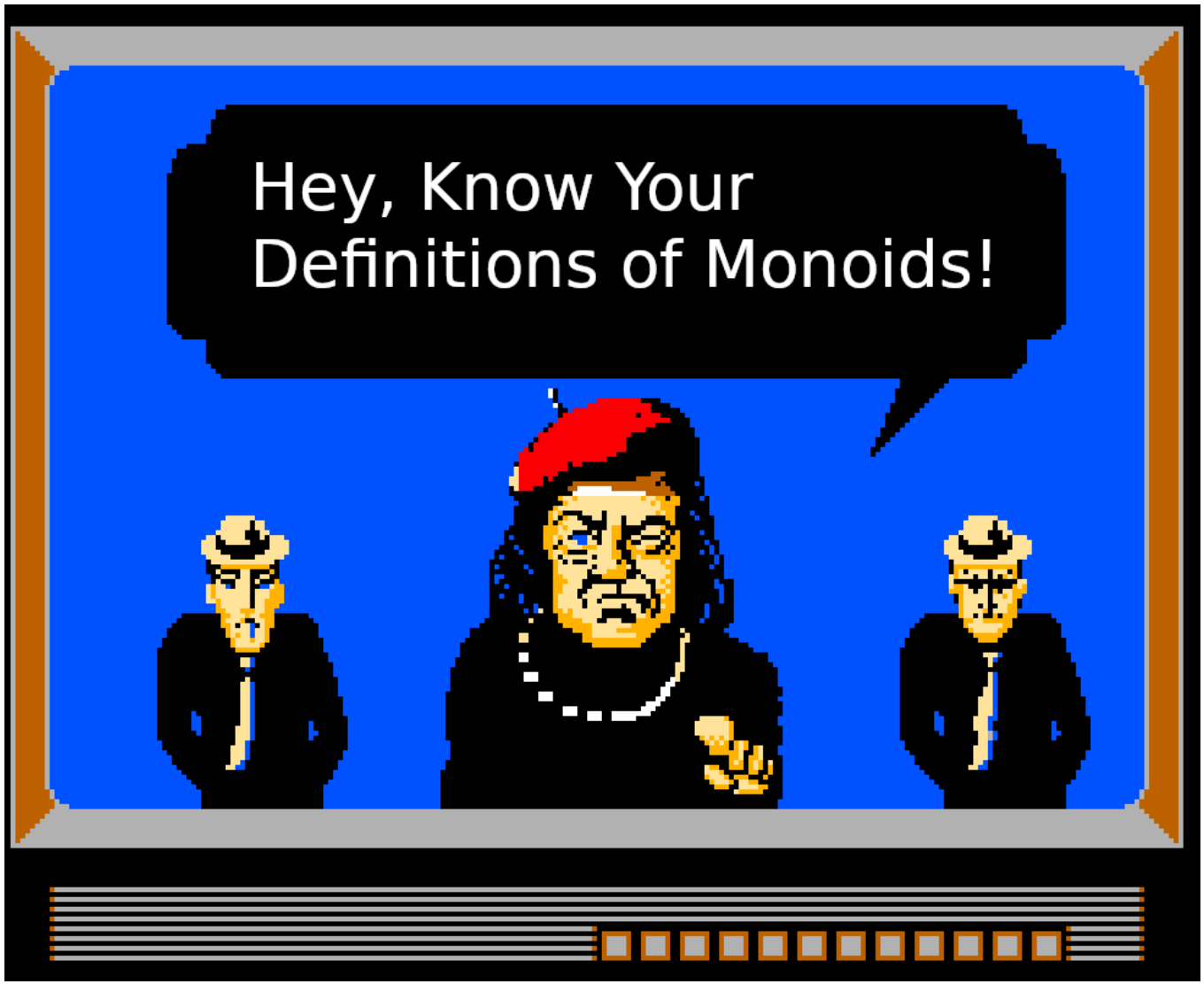
Key Questions

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

And Now This TV



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 $e + a = a + e = a$
    - $n + 0 = n$

Ref: `https://en.wikipedia.org/wiki/Monoid#Definition`

Monoids?

Let's see that in code!

Associative property: Addition with three integers

int + int + int = int

```
a = 10
b = 7
c = 4
print(f"{a+b+c=}")
print(f"{a+b+c==10+7+4}")
```

Identity element: Addition With Null Object

int + Null = int

```
a = 10
b = 0
print(f"{a+b=}")
print(f"{a+b==10+0}")
```

Examples of Sequences in Python

- Sequences are commonly found in Python programs!
- Examples of the **sequence discrete structure** in Python:
  - A string is a sequence of individual characters
  - The `range(20)` function returns a sequence of numbers
  - Files are sequences of lines containing content
  - Each line in a file is a sequence of individual characters
  - Each individual character is a sequence of numbers
  - Each individual number is a sequence of binary digits
- Do these sequences all have properties in common?
- Can we **generalize**?

Licensed to Sequence

A demonstration

Make a Sequence

```
first = "James"
last = "Bond"
print(f"The name is, {last}, {first}-{last}")
```

What is an *n*-Tuple

Make an *n*-tuple

```
myStuff = ()
type(myStuff) # is tuple
item_1 = "Omega Watch"
item_2 = "Aston Martin"
item_3 = "Spy Manual"
myStuff = list(myStuff) # conv to list
type(myStuff) # is list
myStuff.append(item_1)
myStuff.append(item_2)
myStuff.append(item_3)
myStuff = tuple(myStuff)
type(myStuff) # is tuple
print(myStuff)
```

What the difference?

What is the difference between **sequences** and **tuples**?

Comparing Lists and n-Tuples

- Lists are mutable, Tuples are not

```
# Lists are mutable
my_list = [1, 2, 3]
my_list[0] = 99
print("List after modification:", my_list)
```

An Example *mutable* lists

```
a = [2,3,5,7,11]
print(a)
a[2] = False
print(a)
```

Comparing Lists and n-Tuples

- Lists are mutable, Tuples are not

An error is raised

```
# Tuples are immutable
my_tuple = (1, 2, 3)
# The following line would raise an error:
#  'tuple' object does not support item assignment
my_tuple[0] = 99
```

```
my_tuple = (1,2,3,4,5,6)
print(f"my_tuple : {my_tuple}, {type(my_tuple)}")
my_tuple[2] = False
```

String Concatenations in Python

$$\underbrace{s_0, s_1, s_2, \ldots, s_n}_{+} \qquad \underbrace{s_0, s_1, s_2, \ldots, s_n}_{\bigotimes}$$

A sequence of operands to be concatenated. (Note $\bigotimes$ is a generalization.)

```
hello = "hello"
world = "world"
space = " "
message = hello + space + world
print(f"The message is: {message}")
```

- You can concatenate or "glue together" strings
- Can we change orders?

  `hello + space + world`, `space + hello + world`, or `world + space + hello`

Reversed String Concatenation

```
hello = "hello"
world = "world"
space = " "
message = world + space + hello
print(f"The message is: {message}")
```

- What is the **output** of this program segment?
- How does Python **represent** a string in memory?
- What are the different **types** of strings?
- What is an **empty string** in Python?
- How is an empty string different from " "?

Licensed to Sequence

Does this have the same sense?!

Make a Sequence

```
first = "James"
last = "Bond"
print(f"The name is, {first}, {last}-{first}")
```

- Are concatentated sequences still monoids?

Empty String Concatenation in Python

```
firstVar = "hello"
secondVar = "world"
empty = "_"
message = firstVar + empty + secondVar
print(f"The message is: {message}")
```

- The ‘empty‘ variable is an identity string
- What is the output of this program segment?
- What if we switched the order of the concatenation?
- How is the ‘empty‘ variable different from “” ”‘ ?
- What is an ”identity content” for other data types and operators?

Reversed Empty String Concatenation

```
firstVar = "hello"
secondVar = "world"
empty = "_"
message = secondVar + empty + firstVar
print(f"The message is: {message}")
```

- What is the output of this program segment?
- Why does the order of operations not matter in this case?
- Can we generalize these observations about strings?
- Can we define a general discrete structure with predictable properties?
- If you get confused, revisit what you know about working with `str`'s in Python!

Characterizing String Concatenations

- Define S to be the set of all possible strings
- What properties of S are always true?
  - For $s_1, s_2 \in S$ and the concatenation operator "+", $s_1 + s_2 \in S$
  - For $s_1, s_2, s_3 \in S$, "+" **is associative**: $(s_1 + s_2) + s_3 = s_1 + (s_2 + s_3)$
  - For $s_1, s_2, \in S$, "+" **is not commutative**: $(s_1 + s_2) \neq s_2 + s_1$
  - For $s_1, s_2, \in S$, if $s_1 = s_2$ or $s_1 = \epsilon$, then "+" **is commutative**
- These properties of strings help us to **generalize** and **understand** their behavior!
- The **monoid** discrete structure generalizes data that "behaves like strings"

Properties (of real numbers)

Said in a different way from previous slide

Property	Addition	Multiplication
Commutative	$a+b=b+a$	$a \cdot b = b \cdot a$
Associative	$a+(b+c)=(a+b)+c$	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$
Distributive	$a \cdot (b+c) = a \cdot b + a \cdot c$	$a \cdot (b+c) = a \cdot b + a \cdot c$
Identity	$a+0=a$	$a \cdot 1 = a$
Inverse	$a+(-a)=0$	$a \cdot \frac{1}{a} = 1$

- Remember that strings do not behave like numbers when using these properties.

Properties of Strings and Integers

String

- Concatenation through the use of the + operator
- Identity: exists in the "" string
  - "this" + "" = "this"
  - `len("this" +"")`
- Concatenation is associative but **is not** commutative

Integers

- Two integers separated by an + operator creates another integer.
- Addition of integers is the associative property.
- Identity: exists as a 0
  - $n + 0 = n$
- Concatenation is associative and commutative

Monoid Classes :: `__init__`

File: `sandbox/base_permutations.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

class Monoid:
    def __init__(self, null, typeify, operator):
        # __init__ allows class variables to be definded
        # when the class is initiated
        self.null = null
        self.typeify = typeify
        self.operator = operator
```

- Sets up the class in terms of object's variables

Monoid Classes :: `__call__`

File: `sandbox/base_permutations.py`

```
def __call__(self, *args):
    # __call__ method enables classes for which
    # the instances behave like functions and
    # can be called as such
    result = self.null
    for arg in args:
        arg = self.typeify(arg)
        result = self.operator(result, arg)
    return result
```

- Sets up ability for the class to be *called* as a function to simplify programming

Main Function :: `cartesian_prod()`

File: `sandbox/base_permutations.py`

```
def cartesian_prod(a_list,b_list):
    print(f"my a_list and my b_list : {a_list} && {b_list}")
    # input()
    c = []
    for a in a_list:
        for b in b_list:
            c.append(a+b)
    return c
```

- Function to Calculate Cartesian product

Command

File: `sandbox/base_permutations.py`

```
cartesian_product_monoid  =
  Monoid([''],
  lambda x: x,
  cartesian_prod)
# define class
```

- Command to initiate class and pass in list variables for permutation calculation

Command

File: `sandbox/base_permutations.py`

```
base_list = ['A','C','G','T']

print("Length 2 cartesian products")
permutations_list = cartesian_product_monoid(base_list, base_list)
print(f"\t [+] Length 2 Permutations_list = {permutations_list}")
print(f"\t [+] Number of permutations : {len(permutations_list)}")
```

- Prepare the list of characters
- Call `cartesian_product_monoid()`, assign all results to `permutations_list for length 2 products`

Command

File: `sandbox/base_permutations.py`

```
print("Length 4 cartesian products")
permutations_list = cartesian_product_monoid(base_list, base_list, base_list, base_list)
print(f"\t [+] Length 4 Permutations_list = {permutations_list}")
print(f"\t [+] Number of permutations : {len(permutations_list)}")
```

- Prepare the list of characters
- Call `cartesian_product_monoid()`, assign all results to `permutations_list for length 4 products`