Discrete Structures: CMPSC 102

Oliver BONHAM-CARTER

Fall 2019
Week 5

Quiz 1

- Given on Friday 27th during class time (11am)
- Online format
- One hour to complete
- Ten questions: Multi-choice, True/False, Matching and Short Essay
- Picking out bugs of code or determining output

What to study

- **Slides, notes, with chapters to add detail to class material**
- Main ideas behind mathematical subjects in class (again, study your slides)
- Python basics and code
  - Study the code from the practicals and covered in class to understand the how programs worked.
  - Mathematical operators: using doing calculations on in the interpreter with Python
  - `for` loops using `range()`
  - Iterations over sequences
  - Strings, characters, integers, floats
  - Sequences, sets, lists, dictionaries, tuples
  - Conditional statements

Dictionaries

Nomenclature

```
myDict = {key_1:value, key_2:value,... , key_n:value}
```

Example 1

```
myDict = {1:"one", 2:"two", 3:"three"}
```

- Dictionaries work like sets in that there is no repetition in the *keys*
- Values can repeat, as long as the key is unique.

Example 2

```
# this is allowed
myDict = {1:"one", 2:"one", 3:"one"}
# nope: values are over-written of non-unique keys
myDict = {1:"one", 1:"two", 1:"three"}
```

Dictionaries

Defining and Adding to Dictionaries

```
myDict = {} # defined like a set (sort-of)
myDict[0]="zero"
myDict[1]="one"
myDict["roses"]="Red"
myDict["favNum"]=13
print(myDict)
```

Using Keys and Values

```
print("\t My favorite number is :",myDict["favNum"])
print("\t My roses are this color :",myDict["roses"])
```

Removing keys and values

```
del myDict[0] # lose the zero... and get with the hero?
print(myDict[0])
```

Working with Dictionaries

sandbox/dictionaryDemo.py

An example

```
# Date: 23 Sept 2019
# Dictionary Demo by OBC

print("\t You will be asked for three strings")
print("\t which will be stored in a dictionary")
num_dict  = {}
prompt = "Enter a number :"
for i in range(3): # get three numbers
   x_str = input(prompt)
   num_dict[i] = x_str

print("\t Dictionary data structure: ",num_dict)
print("\t + First string: ",num_dict[0])
print("\t + First string: ",num_dict[1])
print("\t + First string: ",num_dict[2])
```

Iterating Over Types of Sequences

- Strings, which are sequences of characters.
- Files contain a sequence of lines and the lines are sequences of characters.
- Objects, over which the `range()` function, can iterate

Examples

```
for element in [1, 2, 3]: # lists
    print(element)
for element in (1, 2, 3): #sets
    print(element)
for key in {'one':1, 'two':2}: #dictionaries
    print(key)
for char in "123": #strings
    print(char)
for line in open("myfile.txt"): # open, read a file
    print(line, end='') #no "\n" printed
```

Building Tuples

To cover this again...

Building Tuples in Python

```
# Creating non-empty tuples
myTuple = 'tea', 'coffee'
print(myTuple)
print(type(myTuple))
```

Or, Use Parenthesis to Build Tuples in Python

```
myOtherTuple = ('Bagels', 'Donuts')
print(myOtherTuple)
print(type(myOtherTuple))
```

Tuples and n-Tuples

Mathematically Speaking...

- An *ordered list* has related items of some defined order
- A *tuple* is an immutable, finite ordered list (sequence) of elements
- An n-tuple is a sequence (or an ordered list) of n elements (n is a positive integer).
  - Ex: (2, 7, 4, 1, 7) denotes a 5-tuple.

General Rule About Equality

- The general rule for the identity of two n-tuples is

$$(a_1, a_2, ..., a_n) = (b_1, b_2, ..., b_n) \text{ if and only if } a_1 = b_1, a_2 = b_2, ..., a_n = b_n$$

```
a = (1,2,3)
b = (1,2,3)
a == b # True
b = (1,3,2)
a == b # False
```

Tuples and n-Tuples
Equality

- A tuple may contain multiple instances of the same element,
  tuple $(1, 2, 2, 3) \neq (1, 2, 3)$ but,
  set $(\{1, 2, 2, 3\}) = \{1, 2, 3\}$
- Tuple elements are ordered,
  tuple $(1, 2, 3) \neq (3, 2, 1)$ but,
  set $(\{1, 2, 3\}) = \{3, 2, 1\}$
- A **tuples**, immutable objects, have a finite number of elements (also known as n-tuples), while sets or **multisets** may have an **infinite** number of elements. (How could a set be infinite?!)

```
a = (1,2,3)
s = set({1,2,3})
a == s # False
```

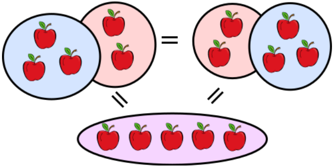
Elements of Tuples

- Sequences are not generic: they usually contain similar types of elements.
  - Ex: Lists contain same types of data structures, strings contain chars, files contain lines
- Sequences and n-tuples
  - n-tuples: An ordered set with n elements
    - Ex: File sequences are not n-tuples because they can contain any number of lines

General Properties of Real Numbers

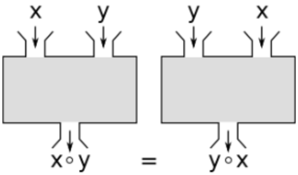
Property	Addition	Multiplication
Commutative	$a + b = b + a$	$a \cdot b = b \cdot a$
Associative	$a + (b + c) = (a + b) + c$	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$
Distributive	$a \cdot (b + c) = a \cdot b + a \cdot c$	$a \cdot (b + c) = a \cdot b + a \cdot c$
Identity	$a + 0 = a$	$a \cdot 1 = a$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

- The term "commutative" is used in several related senses.
- A binary operation $*$ on a set S is called *commutative* if: $x * y = y * x$ for all $x, y \in S$
  - An operation that does not satisfy the above property is called *non-commutative*.
- One says that x *commutes* with y under $*$ if: $x * y = y * x$
- A binary function $f : A \times A \to B$ is called *commutative* if: $f(x, y) = f(y, x)$ for all $x, y \in A$



Commutative

- The operator each side of equation do not create inequality
- Think operators like: Addition, multiplication, division



Not Commutative

- The operator each side of equation creates inequality
- Think operators like: subtraction
- $x - y \neq y - x$; $5 - 3 \neq 3 - 5$

Properties

Non-Commutative operations

- Washing and drying clothes resembles a noncommutative operation; washing and then drying produces a markedly different result to drying and then washing.
- Putting on left and then right socks on feet is commutative
- Putting on shirt and then sweater is not-commutative

Strings

```
a = "face"
b = "book"
a + b == b + a # run the test!
"facebook" != "bookface"
```

Properties

Formal Definition of Identity

- **Identity**: There exists an element $e \in S$ such that for any $a \in M$, $e * a = a * e = a$

Identity

- An identity is an equality relation $a = b$,
- Ex: a and b equal some numeric value.
  - a + b == a + b
  - a + b == b + a
  - a * b == a * b
  - a * b == b * a

- $a = a + e$
- $a + e = a$
- a is non-empty, contains some element
- e must be an empty sequence or is equal to 0
  - e has an *identity* property, meaning that it does not influence the operations
- $a * e = a$ or $a = a * e$, (what is e, the identity here?)

Properties

Identity

Additive Identity

$$a + (0) = a$$
$$0 + (a) = a$$

Remember: Zero (0) preserves the Identity of every number during addition.

```
a = 1
b = 0
a == a + b #make a truth test
a + b == a #make another truth test
```

Properties

Associative Property

Formal Definition of Associativity

- **Associativity Addition**: For any $a, b, c \in S, a + (b + c) = (a + b) + c$
- **Associativity Multiplication**: For any $a, b, c \in S, a * (b * c) = (a * b) * c$

Properties

Associative Property

- Definition: The associative property states that you can add or multiply regardless of how the numbers are grouped.
- Concatenation of sequences with the associative property

- $(a + b) + c = a + (b + c)$ for any strings a, b and c.

```
a, b, c = 1, 2, 3
(a + b) + c == a + (b + c)
```

- $(a * b) * c = a * (b * c)$ for any strings a, b and c.

```
a, b, c = 1, 2, 3
(a * b) * c == a * (b * c)
```

Generalized Associative Law: Keep variables in same order

- ((ab)c)d
- (ab)(cd)
- (a(bc))d
- a((bc)d)
- a(b(cd))

To Note:

- **Associative**: Variables kept in same order, operators may change order
- **Commutative**: Variables may change order, operators kept in same order.

- Definition: a series of interconnected things or events. The concatenation is to place one string after another. The order of placement is significant to the final product.

Ex: Concatenation of sequences

```
a = ("This", "Is")
type(a)
b = ('Loads', 'Of', 'Fun', ':-)')
type(b)
c = a + b
print(c)
(  'This', 'Is', 'Loads', 'Of', 'Fun', ':-)'   )
type(c)
```

Let's Apply Sequences

Need to install a plotting library...

matplotlib
Version 3.1.1

- Matplotlib is library for Python that includes a statistical tools and plotting
- Web: `https://matplotlib.org`

MatPlotLib install using pip

```
pip install --user matplotlib
```

Now, let's get coding using some of the sequence properties that we just talked about!

Let's Apply Sequences
Modelling Interest Rates

Problem:

Put x_0 money in a bank at year 0. What is the value after N years if the interest rate is p percent per year?

Solution:

The fundamental information relates the value at year n, x_n to the value of the previous year, x_{n-1}.

$$x_n = x_{n-1} + \frac{p}{100} * x_{n-1}$$

Start with x_0 and then calculate x_1, then x_2, and onward...

The output of the program?

- $x_0 = 100000$ # initial amount
- $p = 3.92$ # interest rate
- $N = 6$ # number of years

```
At year =  0 Current value is =  103920.0
At year =  1 Current value is =  107993.664
At year =  2 Current value is =  112227.0156288
At year =  3 Current value is =  116626.3146144896
At year =  4 Current value is =  121198.06617539376
At year =  5 Current value is =  125949.0303694692
```

Test these values online

- For example: `http://www.moneychimp.com/calculator/compound_interest_calculator.htm`

Monoids

Monoid: Stavely's Definition, Section 6.2, pp. 59

Both strings with concatenation and integers with addition are examples of the mathematical structure called a *monoid*. A monoid is a set that has an associative binary operator and an identity element.

More formally, a *monoid* is an ordered pair $(S, \bigotimes)$ such that S is a set and $\bigotimes$ is some **binary operator**, satisfying these conditions:

1. For all a and b in S, $a \bigotimes b$ is defined and is also in S
2. For all a, b and c in S, $(a \bigotimes b) \bigotimes c = a \bigotimes (b \bigotimes c)$
3. There is an element e in S such that, for all a in S, $e \bigotimes a = a \bigotimes e = a$
4. Then we also say that S is a *monoid* under $\bigotimes$, with identity e

Monoids

Rounding errors

- The Python floating-point numbers are not quite a monoid under addition: for floating-point operands, $(a + b) + c$ is often not exactly equal to $a + (b + c)$ because of the error during round-off processes
- The same is true of multiplication.

Values round to 1, or do they?

```
a = 0.9999999999999999
b = 0.99999999999999999
a == b #False
```

Min Function

- The Min function, $min(x, y)$, is defined to be x if $x \leq y$ and y otherwise.
- We treat min as an operator and so, $x\ min\ y$ is an operator (like the *max* operator)
- Here, min is both associative and commutative, and the identity value is obtained from `float("inf")` (an inferior value)

Max Function

- The Max function, $max(x, y)$, is defined to be x if $x \geq y$ and y otherwise.
- We treat max as an operator and so, $x\ max\ y$ is an operator
- Here, *max* is both associative and commutative, and the non-negative integers are a monoid under *max*, with identity 0

Monoids

Big Plus and Big Times

$$\underbrace{s_0 \quad s_1 \quad s_2 \quad \cdots \quad s_n}_{+} \qquad \underbrace{s_0 \quad s_1 \quad s_2 \quad \cdots \quad s_n}_{\otimes}$$

Figure: The operator '$+$' is associative, the operator $\otimes$ behaves as an associative

- Knowing that one type of calculation is monoid allows us to use monoid-type code on it.