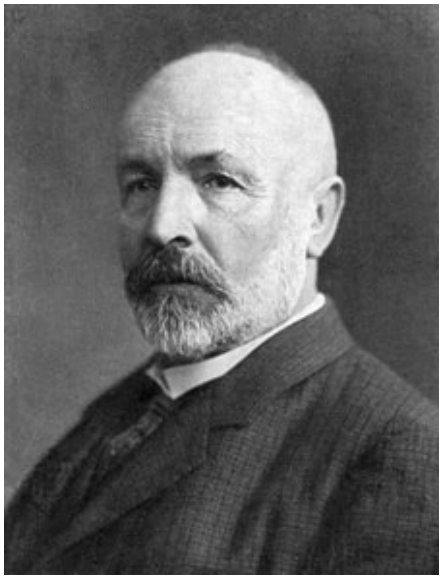
Discrete Structures: CMPSC 102

Oliver BONHAM-CARTER

Fall 2018
Week 4

Georg Ferdinand Ludwig Philipp Cantor

Creator of Set theory



- German mathematician: 19 February 1845 - 6 January 1918
- Function definition: established the importance of one-to-one correspondence between the members of two sets
- Defined infinite and well-ordered sets
- Proved that the real numbers (*rational* and *irrational*) are more numerous than the natural numbers (*counting* numbers)

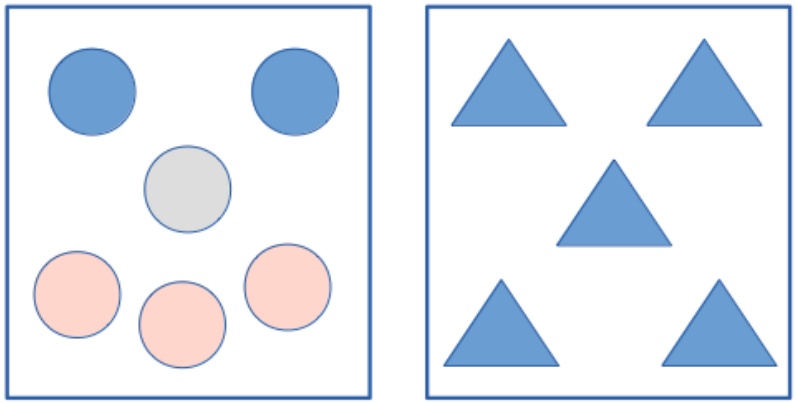
Sets

What is a set?

- A collection of distinct objects is in mathematics, considered to be *an object* in its own right.
  - For example, the numbers 1, 2, and 3 are distinct objects when considered separately, but when they are considered collectively they form a single set of size three, written $\{1,2,3\}$.
- Set theory is now a ubiquitous part of mathematics,
- May be used as a foundation from which nearly all of mathematics can be derived (From 19^{th} century mathematical thinking!)

Types of Sets

One decides which elements make up a set



Intentional definition of sets

- A_1 is the set whose members are the first four positive integers.
- B_1 is the set of colors of the Union Jack (i.e., the British flag)

Types of Sets

Sets of members in curly brackets

Extensional definition of sets

- $A_2 = \{4, 2, 1, 3\}$
  - The first four positive numbers
- $B_2 = \{\text{Blue, Red and White}\}$
  - The set of colors of the Union Jack (the British flag)
- $F = \{n^2 - 4 : n \text{ is an integer; and } 0 \leq n \leq 19\}$
  - The set of all values gained from plugging in n between 0 and 19 into the equation $n^2 - 4$

Types of Sets

Extensional definition of sets: a list of its members in curly brackets

- **Intentional Definition:**
  - A_1 is the set are the first four positive integers.
  - B_1 is the set of colors of the Union Jack
- **Extensional Definition:**
  - $A_2 = \{4, 2, 1, 3\}$
  - $B_2 = \{\text{Blue, Red and White}\}$

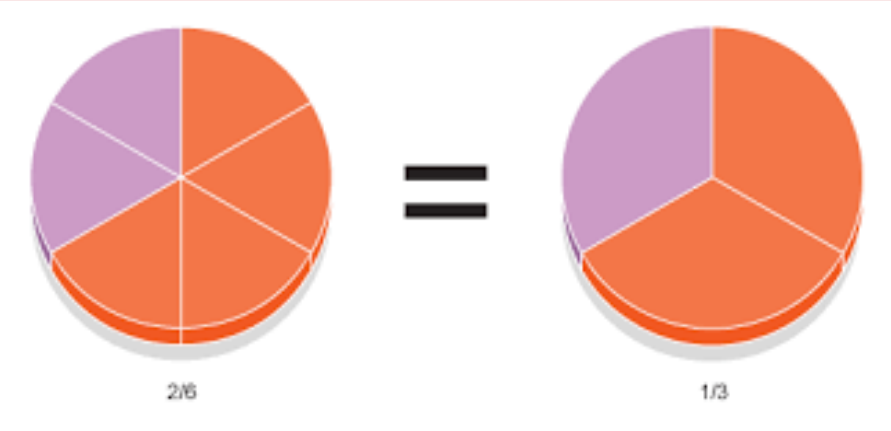
Specify a set *intensionally* or *extensionally*

In the examples above, for instance, $A_1 = A_2$ and $B_1 = B_2$

Listing Elements in Sets

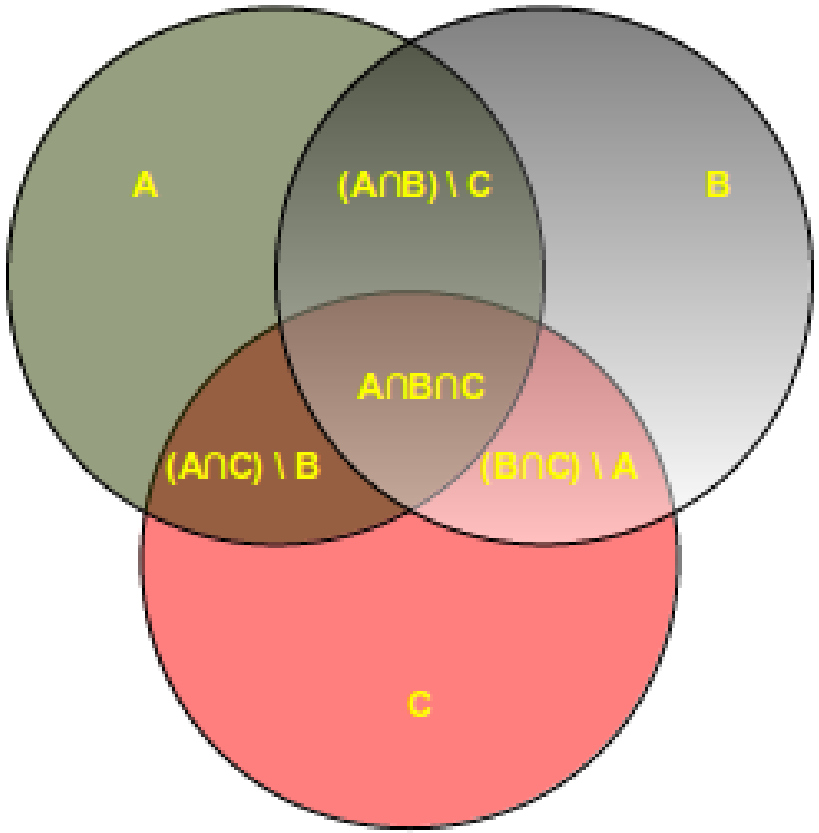
- In extensionally defined sets, members in braces can be listed two or more times,
  - For example, $\{11, 6, 6\}$ is identical to the set $\{11, 6\}$
- Order of members is not important
  - For example, $\{6, 11\} = \{11, 6\} = \{11, 6, 6, 11\}$

Similar to the equivalence of these pie charts:
the content is the same in both cases



Sets with Notation

Venn Diagram



- $\cup$, Union: $A \cup B$ of a collection of sets A and B is the set of all elements in the collection
- $\cap$, Intersection $A \cap B$ of two sets A and B is the set that contains all elements of A that also belong to B

Sets in Python

An array of non-redundant elements

Creating a set of chars

```
x_st = set("This is a set")
x_st   # or print(x_st)
   # the unordered chars are the elements
   # {'s', 'T', ' ', 'e', 't', 'h', 'i', 'a'}
print(type(x_st))
   # <class 'set'>
```

Creating a set of string(s)

```
x_st = set(["This is a set"])
x_st   # or print(x_st)
   # only one element in set; the string itself
   #{'This is a set'}
x_st = set(["This", "is", "a", "set"])
   # each word is an element
   #{'This', 'is', 'set', 'a'}
```

Sets in Python


```
# next line on one line
cities_st = set(("Paris", "Lyon",
     "London","Berlin","Birmingham", "Paris"))
print(cities_st)
   # {'Berlin', 'Paris', 'Birmingham', 'London', 'Lyon'}
```

Adding new elements

```
cities_st = set(["Frankfurt", "Basel", "Freiburg"])
cities_st.add("Meadville")
cities_st # or print(cities_st)
   # {'Freiburg', 'Meadville', 'Basel', 'Frankfurt'}
```

Sets in Python

Removing elements

```
cities_st = set(["Frankfurt", "Basel", "Meadville"])
cities_st.remove("Meadville")   # Meadville is a key
cities_st   # or print(cities_st)
   # {'Basel', 'Frankfurt'}
```

Frozensets cannot be changed

```
cities_st = frozenset(["Frankfurt", "Basel", "Freiburg"])
cities_st.add("Meadville")
   # AttributeError:
   # 'frozenset' object has no attribute 'add'
cities_st # or print(cities_st)
   # frozenset({'Freiburg', 'Basel', 'Frankfurt'})
type(cities_st)
   # <class 'frozenset'>
```

Sets in Python

Removing all elements of set

```
cities_st = {"Stuttgart", "Konstanz", "Freiburg"}
cities_st
   # {'Freiburg', 'Konstanz', 'Stuttgart'}
cities_st.clear()
cities_st
   # set()
```

Determining difference between sets

```
x = {"a","b","c","d","e"}
y = {"b","c"}
z = {"c","d"}
x.difference(y)  # {'a', 'e', 'd'}
x.difference(y).difference(z)  # {'a', 'e'}
```

Sets in Python

Difference and subtraction

```
x = {"a","b","c","d","e"}
y = {"b","c"}
x.difference_update(y)
print(x)  # {'a', 'e', 'd'}
x = {"a","b","c","d","e"}
y = {"b","c"}
x = x - y
print(x)   # {'e', 'd', 'a'}
```

Sets in Python

Cloning and removing from original

```
x = {'e', 'd', 'a'}
v = x
print(x)   # {'a', 'e', 'd'}
print(v)   # {'a', 'e', 'd'}
x.remove('a')
x   #  {'e', 'd'}
v   #  {'e', 'd'}
v.remove('d')
x   # {'e'}
v   # {'e'}
```

$x = v$ does not make a copy of x. Instead this is a reference from one object to another.

Checking for Particular Elements

Subtraction

```
x = {"a","b","c","d","e"}
"e" in x   # True
"e" and "a" in x  # True
"e" and "i" in x  # False
```

Lists in Python

Lists, similar to arrays, are collections which are ordered and changeable.

Creating lists from scratch

```
myList_list = []
myList_list #or print(myList_list)
   # []
myList_list.append("x")
myList_list.append("x")   # again
myList_list   # ['x', 'x']
```

Creating lists in entirety

```
myList_list = ["a","b","c","d"]
myList_list #or print(myList_list)
   #['a', 'b', 'c', 'd']
type(myList_list)
   #<class 'list'>
```

Lists in Python

Removing an element

```
myList_list = ["a"]
print(myList_list)
   #   ['a']
myList_list.remove("a")
print(myList_list)
   #   []
```

Reverse the entire list, no assignment necessary

```
myList_list = ["a","b","c","d"]
myList_list.reverse()
myList_list  #or print(myList_list)
   # ['d', 'c', 'b', 'a']
```

Lists in Python

Each element has a location

```
myList_list = ["a","b","c","d"]
myList_list[0]  # 'a'
myList_list[3]  # 'd'
myList_list[300] #IndexError
```

Print each element by location

```
for i in range(len(myList_list)):
    print("index = ",i)
    print("   myList_list[i] = ",myList_list[i])
#    index =  0
#    myList_list[i] =  a
#       ...
#    index =  3
#    myList_list[i] =  d
```

Lambda Functions

We will use these to create lists ...

Lambda function definition

- The lambda operator or lambda function is a way to create small anonymous functions (i.e. functions without a name), and are *throw-away* functions

General syntax

lambda argument_list: expression

```
g = lambda x: 3*x + 1
g(2) #   7
```

```
sum = lambda x, y : x + y
sum(3,4) #   7
```

List Comprehensions to build lists

List comprehensions definition

- List comprehensions provide a concise way to create lists (or sets)

General syntax

[expression for item in list if conditional]

Make list

```
[i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Assign list to variable

```
b_list = [i for i in range(10)]
type(b_list)
<class 'list'>
```

List Comps and Lambda Functions to build lists

Build a list with an anonymous function

```
g_list = lambda x: list(i for i in range(x))
g_list(4)   #   [0, 1, 2, 3]
myList_list = g_list(4)
myList_list #   [0, 1, 2, 3]
# slicing particular elements
myList_list[0:2] #   [0, 1]
```

Tuples

A Tuple is a collection of Python objects separated by commas

An empty tuple

```
empty_tuple = ()
print (empty_tuple)
type(empty_tuple)   # <class 'tuple'>
```

A non-empty tuple

```
nonEmpty_tuple = ("a","b","c","d")
nonEmpty_tuple[0]   #   'a'
nonEmpty_tuple[len(nonEmpty_tuple)-1]   #   'd'
```

Check to see that elements are in a tuple

```
nonEmpty_tuple  #  ('a', 'b', 'c', 'd', 4, 'Hi')
"Hi" in nonEmpty_tuple   #  True
4 in nonEmpty_tuple   # True
3 in nonEmpty_tuple   # False
```

Tuples

Check to see that elements are in an element at a tuple location

```
nonEmpty_tuple = ("a","b","c","d", 4, "Hi", "My music")
nonEmpty_tuple
   #  ('a', 'b', 'c', 'd', 4, 'Hi', 'My music')
"my" in nonEmpty_tuple   # False
"My" in nonEmpty_tuple   # False

# check to see if detail is in a substring in tuple
"My" in nonEmpty_tuple[6]  # True
```