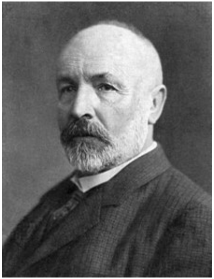
Discrete Structures: CMPSC 102

Oliver BONHAM-CARTER

Fall 2019
Week 4

Georg Ferdinand Ludwig Philipp Cantor

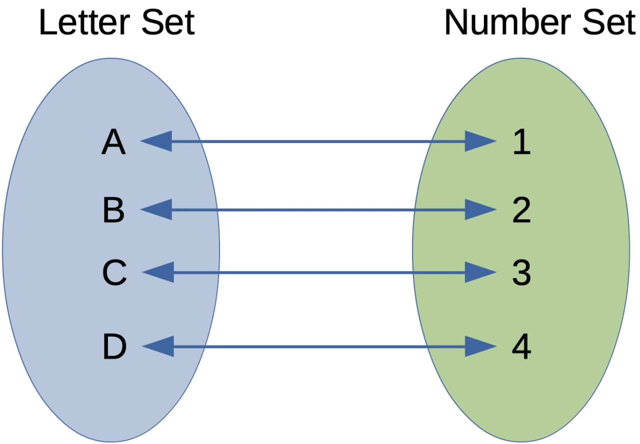
Creator of Set theory



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

Functions as Sets

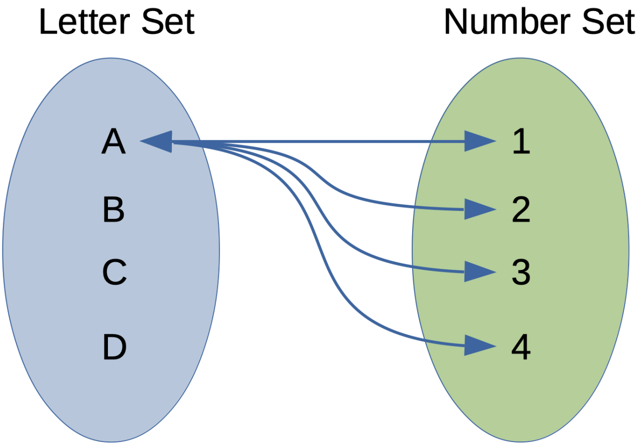
Regular Set: one-to-one relationship maintained



- The Letter set maps to the Number set.
- $LetterSet(x) \rightarrow NumberSet$

Functions Sets

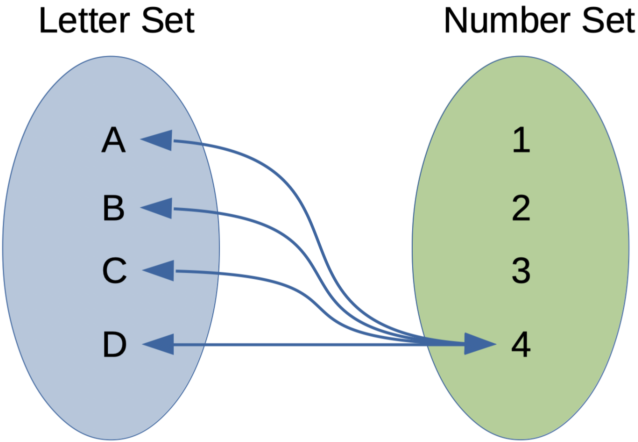
Regular Set: one-to-one-*ism* is maintained



- The Letter set maps to the Number set.
- $LetterSet(x) \rightarrow NumberSet$

Functions as Sets

One-to-one-*ism* is NOT maintained!



- The Letter set maps to the Number set.
- $LetterSet(x) \rightarrow NumberSet$

General Sets

What is a set?

- 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

Intensional and Extensional

Intentional definition of sets: *I intend this set to be ...*

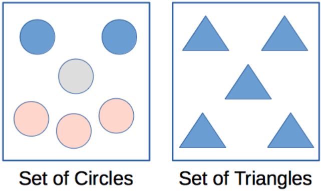
- Defines a set by specifying the necessary and sufficient conditions for when the set should be used.

Extensional definition of sets: *Logically this set is ...*

- Defines a set by some definition of a concept or a term.

Types of Sets

Intentional: One decides which elements make up a set



Intentional definition of sets: *I intend that these set be ...*

- The set of blue, grey and pink circles
- The set of blue triangles
- The set of colors of the Union Jack (i.e., the British flag)

Types of Sets

Extensional: 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

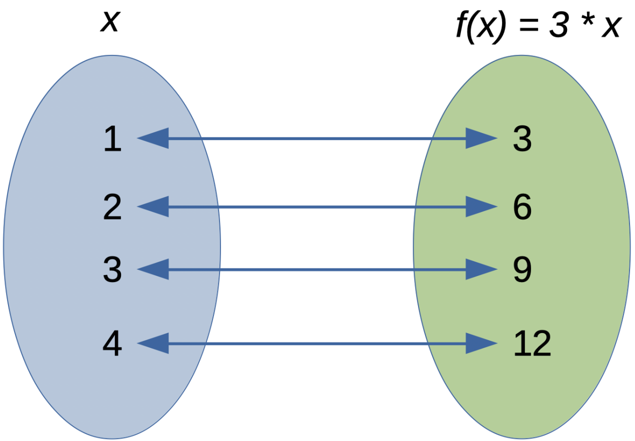
- **Intentional Definition:**
  - A_1 is the set are the first four positive integers.
  - B_1 is the set of colors of the Union Jack
- **Extensional Definition:**
  - $A_2 = \{4, 2, 1, 3\}$
  - $B_2 = \{\text{Blue, Red and White}\}$

Specify a set *intensionally* or *extensionally*

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

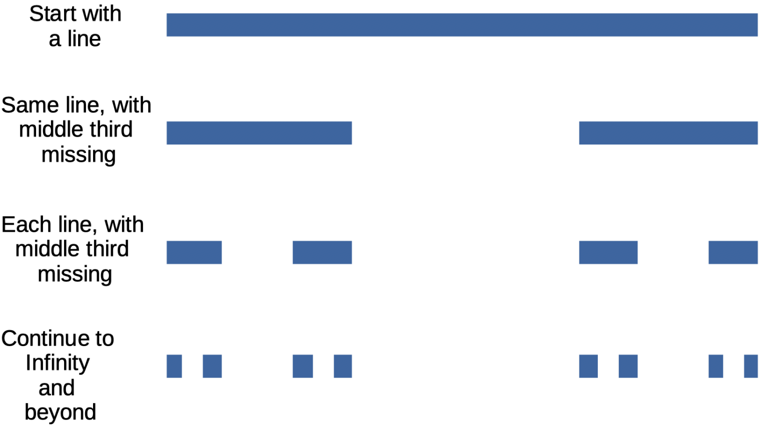
Infinite Sets: an Extensional set example

Sets that go on forever



Infinite Sets: an Extensional set example

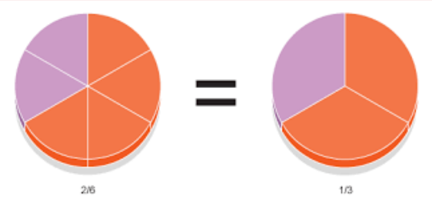
Sets that go on forever

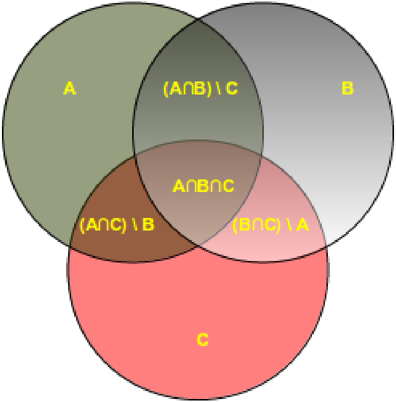


Listing Elements in Sets

- 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





- ∪, Union: $A \cup B$ of a collection of sets A and B is the set of all elements in the collection
- ∩, 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'}
```

- Returns the characters which are never repeated across $\{x, y, y\}$

Sets in Python

Difference and subtraction

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

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

- Top: Returns an updated set of x of the characters which are never repeated across {x, y, y}

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

Is an element in a List?

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

Iterating Through Elements in Sets

Iteration

```
abc_set = {"a","b","c","d","e"}
for i in abc_set:
  print(i)
```

Note

- Since there is no order control in the set, you cannot know which element will be printed first (from above).

Lists in Python

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

Creating Lists with append maintains position information

```
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'>
```

- With a `list`, position of character is maintained, not so with a set.

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
```

Iterating Through Elements in Lists

Iteration

```
l_list = ["a","b","c","d"]
for i in l_list:
  print(i)
```

Iteration

```
l_list = ["a","b","c","d"]
for i in range(len(l_list)):
  print("i = ",i," and l_list[i] = ",l_list[i])
```

Note

- With lists, we know which element will be printed first (the first element, from above).

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
```

Adding to Tuples

Convert tuple to list, add element, convert back

```
a_tuple = ('2',) #define Tuple
items = ['a', 'b', 'c', 'd'] # elements to add
l_list = list(a_tuple)# make a list
for x in items:
   l_list.append(x) # add items to list
#output as a tuple
print(tuple(l_list))
```

Iterating Through Elements in Tuples

Iteration

```
nonEmpty_tuple = ("a","b","c","d", 4, "Hi", "My music")
for i in nonEmpty_tuple:
  print(i)
```

Iteration

```
for i in range(len(nonEmpty_tuple)):
  print("i= ",i, "nonEmpty_tuple[i]=",nonEmpty_tuple[i])
```

Note

- With tuples (like lists), we know which element will be printed first (the first element, from above).

Dictionaries

An array of a key and a value that is connected for quick searching

- A dictionary maps a set of objects (keys) to another set of objects (values).
- A Python dictionary is a mapping of unique keys to values.
- Dictionaries are mutable, which means they can be changed.
- The values that the keys point to can be any Python value

An empty dictionary

```
myDictionary_dict = {}
print (myDictionary_dict)
type(myDictionary_dict)   # <class 'dict'>
```

Dictionaries

Adding to a dictionary

```
myDictionary_dict = {}
myDictionary_dict[0] = "zero"
myDictionary_dict[0] # gives 'zero'

myDictionary_dict[1] = "one"
print (myDictionary_dict)  #{1: 'one', 0: 'zero'}
```

Removing elements from a dictionary

```
myDictionary_dict = {}
myDictionary_dict[3] = "three"

del myDictionary_dict[3]
print (myDictionary_dict)  #{} (is empty)
```

Randomly Choosing Elements

Choosing Elements from a List

```
import random
abc_list = ['a','b','c','d','e']
random.choice(abc_list)     # 'c'
random.choice(abc_list)     # 'd'
```

Choosing Elements from a List

```
import random
abc_set = set(['a','b','c','d','e'])
  # convert to list
abc2_list = list(abc_set)
random.choice(abc2_list)    # 'd'
```

Randomly Choosing Elements

Choosing Elements from a Dictionary

```
import random
abc_dict = {1:"one",2:"two",3:"Three"} # {vals : keys}
num_list = list(abc_dict) # convert dict to list
n = random.choice(num_list) # pick a number in list
abc_dict[n]  # sub in n to get key value
  #  'two'
```

This Week: You will be comparing Lists

import random


```
aliceVocab_list = ["I like cats", "I like dogs",
"I like rabbits", "I gave carrots to horses",
"I live on a farm"]

aliceSays_str = random.choice(aliceVocab_list) # choose rar
print("  This is Alice. I say to Bob :", aliceSays_str)

bobVocab_list = ["I have two cats", "I have three dogs",
"I know several rabbits","I love carrots",
"I love horses","I also live on a farm"]

bobSays_str = random.choice(bobVocab_list)
print("  This is Bob. I reply to Alice :",bobSays_str)
```

Removing Stop-Words

This week's lab: Remove Words from Strings using Lists

```
stopWords_list =["I", "have","know",
"like", "love", " to ", " a "]

  # we remove stop words
  # as they do not add specificity to the strings

def removeStopWords(in_str): # string input
  for s in stopWords_list:
    in_str = in_str.replace(s,"") #word with empty space
  return in_str.strip()  # remove spaces, return.
  #end of removeStopWords()
```

- Find remove the stop-words and compare the lists to find common words.
- When you find the common words between two lists, you have found a contextual link between them.