Introduction to Computational and Algorithmic Thinking

CHAPTER 7 – RANDOM NUMBERS AND OBJECT ORIENTED PROGRAMMING

Announcements

This lecture: Random Numbers and Object Oriented Programming

Reading: Read Chapter 7 of Conery

Acknowledgement: Some of this lecture slides are based on CSE 101 lecture notes by Prof. Kevin McDonald at SBU

Games of Chance

- Many games involve chance of some kind:
  - Card games with drawing cards from a shuffled deck
  - Rolling dice to determine how many places we move a piece on a game board
  - Spinning a wheel to randomly determine an outcome
- We expect these outcomes to be **random** or unbiased – in other words, *unpredictable*
- Computers can be programmed to generate *apparently* “random” sequences of numbers and other quantities for such games and other applications

Games of Chance

- In this lecture we will explore algorithms for generating values that are apparently random and unpredictable
- We say “apparently” because we need to use mathematical formulas to generate sequences of numbers that at the very least appear to be random
- Since we will use an algorithm to generate “random” values, we really can’t say the sequence of values is truly random
- We say instead that a computer generates **pseudorandom n**umbers

Pseudorandom Numbers

- Randomness is a difficult property to quantify
  - Is the list **[3, 7, 1, 4]** more or less random than **[4, 1, 7, 3]**?
- The algorithm that generates pseudorandom numbers is called **pseudorandom number generator**, or PRNG
- The goal is for the algorithm to generate numbers without any kind of apparent predictability
- Python has a built-in capability to generate random values through its **random** module
- To generate a random integer in the range 1-10:

```
import random
num = random.randint(1,10) # 10, not 11!
```

Modular Arithmetic

- The **mod** operator, denoted **%** in Python, will be a key part of generating pseudorandom numbers
- Suppose we wanted to generate a seemingly random sequence of numbers, all in the range 0 through 11
- Let's start with the number 0 and store it in a new list named **t**:

  t = [0]
- One basic formula for generating numbers involves:

  (1) adding a value to the previously-generated number and then

  (2) performing a modulo operation

Modular Arithmetic

- For our particular example, we could use 7 as our added value and then mod by 12
- Conveniently, the Python language lets us write **t[-1]** to mean "retrieve the last element of list **t**"
- We can write **t[-2]** to get the second-to-last element,
- **t[-3]** to get the third-to-last element, and so on
- So in general we can write **t.append((t[-1]+7)%12)** to generate and store the "next" pseudorandom number
- If we put this code inside a loop, we can generate a series of random values and store them in the list

Modular Arithmetic

```
t = [0]
for i in range(15):
   t.append((t[-1] + 7) % 12)
```

- The above code will generate the list:

  [0,7,2,9,4,11,6,1,8,3,10,5,0,7,2,9]
- How “random” are these numbers?
  - They look pretty random, but we notice that eventually they start to repeat
- Can we improve things?
  - Part of the issue is the divisor of 12, but the formula itself is a little too simplistic

Modular Arithmetic

- A more general formula for generating pseudorandom numbers is $x_{i+1}=(a*x_i+c) \bmod m$
- X_{i+1} is the “next” random number
- X_i is the most recently generated random number
- *i* is the position of the number in the list
- *a*, *c* and *m* are constants called the *multiplier*, *increment*, and *modulus*, respectively
- If the values *a*, *c* and *m* are chosen carefully, then every value from 0 through *m-1* will appear in the list exactly once before the sequence repeats
- The number of items in the repetitive part of the list is called the **period** of the list

Modular Arithmetic

- We want the period to be as long as possible to make the numbers as unpredictable as possible

- We will implement the above formula, but first we need to explore some new programming concepts

Numbers on Demand

- One possibility for working with random numbers is to generate as many as we need and store them in a list
  - Often, however, in real applications we don't know exactly how many random numbers we will ultimately need
  - Also, in practice we might not want to generate a very long list of random numbers and store them
- Typically, we need only one or just a few random numbers at a time, so generating thousands or even millions of them at once is a waste of time and memory
- Rather than building such a list, we can instead generate the numbers one at a time, *on demand*

Numbers on Demand

- We will define a function **rand()** and a **global variable x** to store the most recently generated random number
  - A *global variable* is a variable defined outside functions and is available for use by any function in a .py file
- The value of a global variable is preserved between function calls, unlike local variables, which disappear when a function returns
- If we want a function to change the value of a global variable, we need to indicate this by using the **global** keyword in the function
- If we are only reading the global variable, we do not need to use the **global** keyword

The rand() Function (v1)

- Let’s consider a function for generating random numbers that uses the formula we saw earlier:

```
x = 0    # global variable
def rand(a, c, m):
    global x
    x = (a * x + c) % m
    return x
```

- Call the function several times with a=1, c=7, m=12:

```
rand(1, 7, 12)    # returns 7 and updates x to 7
rand(1, 7, 12)    # returns 2 and updates x to 2
rand(1, 7, 12)    # returns 9 and updates x to 9
```

- Let’s see why x is updated in this way

The rand() Function (v1)

- The key line of code is **x = (a * x + c) % m**
- Initially, **x = 0**
  1. **rand(1,7,12): x = (1 * 0 + 7) % 12 = 7**
     - So, **x** becomes **7**
  2. **rand(1,7,12): x = (1 * 7 + 7) % 12 = 2**
     - So, **x** becomes **2**
  3. **rand(1,7,12): x = (1 * 2 + 7) % 12 = 9**
     - So, **x** becomes **9**
- The only reason this series of computations works correctly is because the value of **x** is preserved between function calls

Modules and Encapsulation

- Suppose we wanted to use our new **rand()** function in several files. We have two options:
  - Copy and paste the function to each file (bad idea), or
  - Place the file in a .py by itself (or with other functions) to create a **module** that can be imported using an **import** statement (the right way)
- We should place our function in a module, along with the global variable **x**
- This global variable will be "hidden" inside the module so that there is no danger of a "name clash", meaning that other modules could have their own global variables named **x** if they want to

Modules and Encapsulation

- This idea of gathering functions and their related data values (variables) into a single package is called ***encapsulation***
- It's an extension of the concept called **abstraction** we studied earlier in the course
- We know that the **math** module has some useful functions and constants, like **`sqrt()`** and **`pi`**
- A module like **math** is an example of a ***namespace***, a collection of names that could be names of functions, objects or anything else in Python that has a name
  - A module/namespace is one way of implementing the concept of encapsulation in Python

Modules and Encapsulation

- To create a new module, all we need to do is save the functions and variables of the module in a file ending in .py
  - For example, if we were to save the **rand()** function in the file **`prng.py`**, we could then import the **rand()** function in a new Python program by typing **`import prng`** at the top of the new program
- Next slide shows a revised version of our **`rand()`** function that encapsulates the function in a module and stores the values of *x*, *a*, *c* and *m* as global variables
- This means the user no longer needs to pass *a*, *c,* or *m* as arguments anymore
- We will also add a new function **`reset()`** to reset the PRNG to its starting state

The rand() Function (v2)

```
x = 0
a = 81
c = 337
m = 1000
```

```
def rand():
   global x
   x = (a * x + c) % m
   return x
```

```
def reset(mult, inc, mod):
   global x, a, c, m
   x = 0
   a = mult
   c = inc
   m = mod
```

The rand() Function (v2)

x = 0

a = 81

c = 337

m = 1000

- Examples:

rand(): (81 * 0 + 337) % 1000 = 337

rand(): (81 * 337 + 337) % 1000 = 634

rand(): (81 * 634 + 337) % 1000 = 691

The rand() Function (v2)

- We can change the values of *a*, *c,* and *m* by calling the **reset()**
- function. Example: **reset(19, 4, 999)**, which also sets **x = 0**.

- Now we will generate a different sequence of random numbers:
  1. **rand()**: **(19 * 0 + 4) % 999 = 4**
  2. **rand()**: **(19 * 4 + 4) % 999 = 80**
  3. **rand()**: **(19 * 80 + 4) % 999 = 525**

Games with Random Numbers

- Suppose we wanted to simulate the rolling of a six-sided die in a board game
- We would want to generate integers in the range 1 through 6, inclusive
- Our function **rand()** generates values outside this range, however
- We can solve this problem using an expression like **rand() % 6 + 1**
- The expression **rand() % 6** gives us a value in the range 0 through 5, which we can then "shift up" by adding 1
- Why not do **rand() % 7** instead?

Games with Random Numbers

- If we always initialize *x*, *a*, *c,* and *m* to the same values, then every program that uses the **rand()** function will get the same exactly sequence of pseudorandom values

Instead, we could allow someone using our code to set the starting value of *x*, which we call the **seed** of the pseudorandom number generator

Another option is we can have the computer pick the seed by using the system clock

The time module has a function called **time()** which returns the number of seconds since January 1, 1970

Fractions of a second are also included in the returned value

Games with Random Numbers

- Our revised module shown on the right uses **time.time()** to pick a random seed

```
import time
a = 81
c = 337
m = 1000
x = int(time.time()) % m
def rand():
   global x
   x = (a * x + c) % m
   return x
```

- See random_numbers.py

Random Numbers in a Range

- In general, how can we generate random integers from an arbitrary range?
- The formula is surprisingly simple:

  rand() % (high – low + 1) + low
- For example, suppose we wanted to generate a value in the range -1 through 10, inclusive
- The formula indicates we should use this code:

  rand() % (10 - (-5) + 1) + (-5)
- Simplifying gives us: **rand() % 16 – 5**

- See random_numbers.py

List Comprehensions

- Python features a very compact syntax for generating a list called a **list comprehension**
  - We write a pair of square brackets and inside the brackets put an expression that describes each list item
- For example, to make a list of numbers from 1 to 10 write **[i for i in range(1,11)]**
- To make a list of the first 10 perfect squares we could write **[i**2 for i in range(1,11)]**
- In general, we write an expression that describes each new item in the new list and a loop that describes a set of existing values to work from
- A list of 10 random numbers:

  [rand() for i in range(10)]

List Comprehensions

- Suppose we wanted to take a list of words and capitalize them all:

```
names = ['bob', 'DANE', 'mikey', 'ToMmY']
names = [s.capitalize() for s in names]
```

- **names** would become **['Bob', 'Dane', 'Mikey', 'Tommy’]**

- Or perhaps we wanted to extract the first initial of each person and capitalize it:

```
initials = [s[0].upper() for s in names]
```

- **initials** would be **['B', 'D', 'M', 'T’]**

Random Shuffles

- Suppose we needed the ability to randomly *permute* (shuffle) a list of items, such as a deck of 52 playing cards
- Let's explore how we might write a function that does exactly this
- The **RandomLab** module defines a **class** called **Card**
- A ***class*** defines a new type of object in an object-oriented programming language like Python
- We use a special method called the **constructor** to create (construct) new objects of the class

```
from PythonLabs.RandomLab import Card
card = Card()
```

The Card Class

- A **Card** object has a separate **rank** and **suit**, which we can query using the **rank()** and **suit()** methods, respectively
- The 2 through Ace are ranked 0 through 12
- The suits are mapped to integers as follows:
  - Clubs: 0
  - Diamonds: 1
  - Hearts: 2
  - Spades: 3
- For example, for a **Card** object representing the 9 of Spades, **rank()** would return 7 and **suit()** would return 3

The Card Class

- The ranks and suits are numbered so that we can uniquely identify each card of a standard 52-card deck
  - When calling the constructor to create a **Card** object, we provide a number in the range 0 through 51 to identify which card we want
- Examples:
  - **Card(0)** and **Card(1)** are the 2 and 3 of Clubs, respectively
  - **Card(50)** and **Card(51)** are the King and Ace of Spades, respectively
  - **Card(46)** is 9 of Spades
- **print(Card(51))** would output **A♠** (yes, including that Spade symbol!)

The Card Class

- We can use a list comprehension to generate all 52 cards and store them in a list:

```
deck = [Card(i) for i in range(0,52)]
```

- With slicing we can take a look at just the first 5 by appending **[:5]** to the name of the variable
- This notation means "slice out all the elements of the list up to (but not including) the element at index 5"

```
print(deck[:5])
```

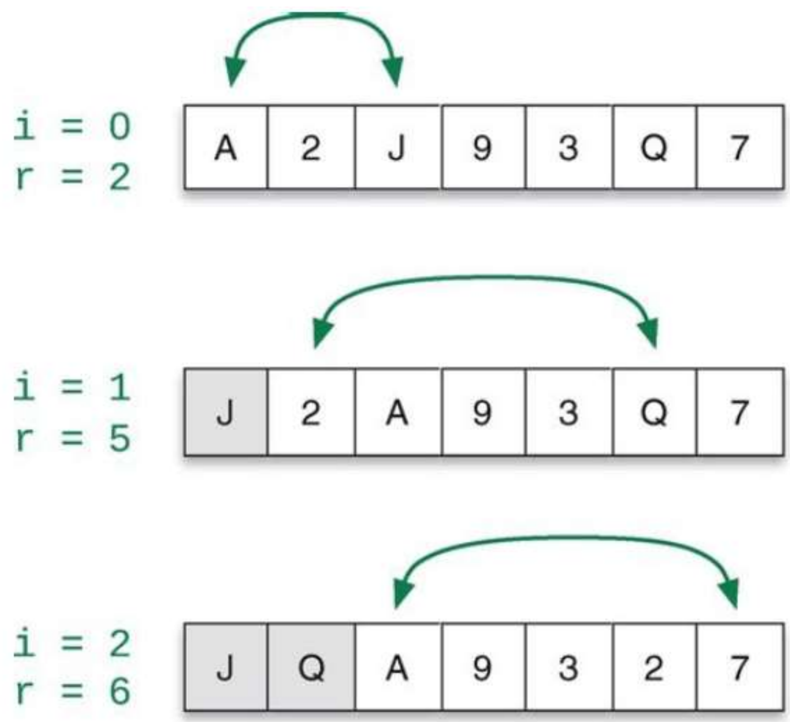
- Output: **[2♣, 3♣, 4♣, 5♣, 6♣]**

Shuffling Card Objects

- The order of the cards generated by the list comprehension (i.e., sequential order) is only one particular ordering or **permutation** of the cards
- We want to define a function that will let us permute a list to generate a more random ordering of the items in the list
- A simple algorithm for permuting the items in a list is to iterate over the list and exchange each element with a random element to its right
- This is most easily seen by example, as on the next slide

Shuffling Card Objects

Iterate over the entire list **deck** (with **i** as the loop variable and index), swapping a random item to the right of **i** with **deck[i]**



Shuffling Card Objects

- This shuffling algorithm is very easy to implement with the help of a function that will choose a random item to the right of **deck[i]**
- The function **randint(low, high)** from the **random** module generates a random integer in the range **low** through **high** (inclusive of both **low** and **high**)
- The **permute** function will shuffle any list of items:

```
import random
def permute(a):
    for i in range(0, len(a)-1):
        r = random.randint(i, len(a)-1)
        a[i], a[r] = a[r], a[i] # swap items
```

Shuffling Card Objects

```
import random
def permute(a):
   for i in range(0, len(a)-1):
      r = random.randint(i, len(a)-1)
      a[i], a[r] = a[r], a[i] # swap items
```

- **r = random.randint(i, len(a)-1)** picks the random index, **r**, that is to the right of **i** (or might choose **i** itself, meaning that **a[i]** doesn't move)
- **a[i], a[r] = a[r], a[i]** swaps **a[i]** with the randomly chosen item to its right
- We would call this function with **permute(deck)** to shuffle our list of **Card** objects

Defining New Objects

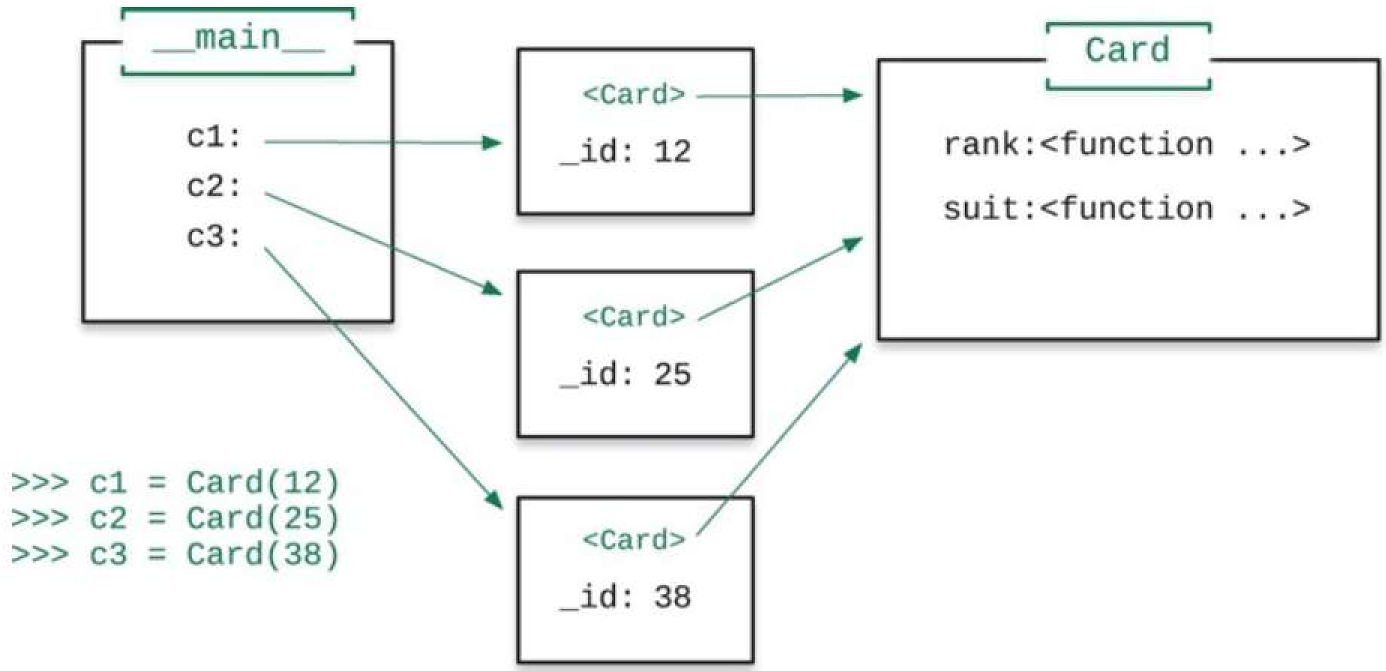
- The **Card** class we have been working with defines a new kind of object we can use in programs
- In object-oriented programming, a class determines the data and operations associated with an object
- For example, for a playing card object we need some way to store the rank and suit of a card; these are its data attributes
- Operations for a playing card might include code that lets us print a playing card on the screen or retrieve the card’s rank and suit

Defining New Objects

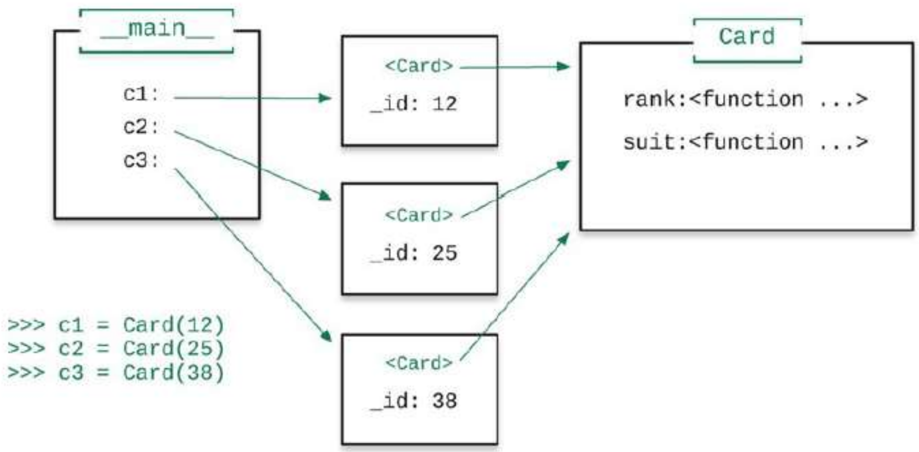
- The data values associated with a particular object are called **instance variables**
- We say that an object is an **instance** of a class
  - For example, each of the 52 **Card** objects is an independent instance of the **Card** class
  - As such, each **Card** object has its own copies of the instance variable that store the object's rank and suit
- The operations associated with an object are called **methods**
- So, a class defines the data properties and methods that an object of the class has

Defining New Objects

Let’s see an example where we create three distinct **Card** objects in a program:

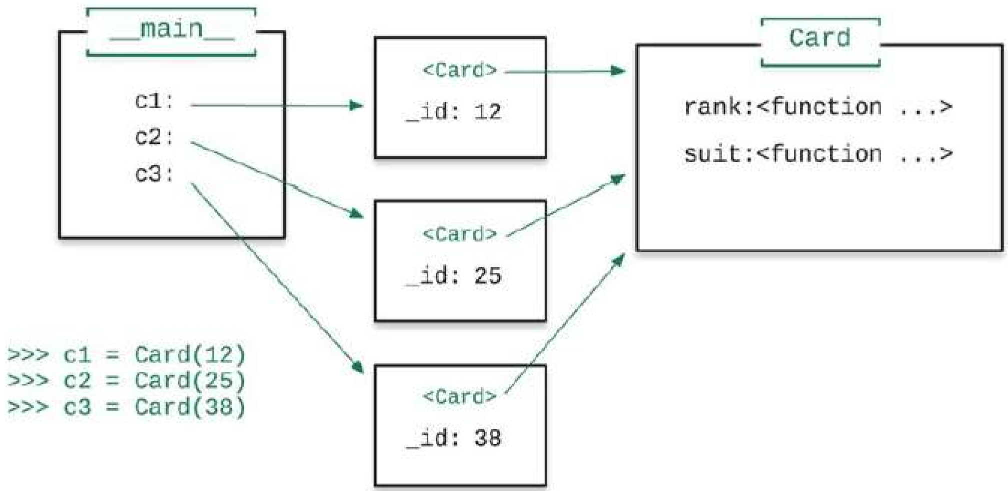


Defining New Objects



- Three **Card** objects were constructed. They are referenced using the variables **c1**, **c2** and **c3** in **main** as shown on the left
- The objects as they might exist in the computer memory are shown in the middle of the diagram
- Rather than storing the rank and suit separately, they are combined into a single integer called **_id**

Defining New Objects



- Prepending an underscore to a variable indicates that **_id** is an instance variable; this is a naming convention, not a strict rule
- To retrieve the rank or suit, we need to call the methods **rank()** or **suit(),** as depicted on the right
- Example call: **c1.rank()** since **rank()** is a method, not a function

Defining New Objects

- To define a new class we usually include the following aspects:
  - One or more instance variables
  - One or more methods that perform some operation or execute some algorithm
  - A `__init__` method, which initializes (gives starting values to) the instance variables
  - A `__repr__` method, which defines a string representation of an object that is suitable for printing on the screen

- Let's step through building the **Card** class from the ground up

Building the Card Class

- The code we build up in piecemeal fashion will all eventually be saved in a file named **Card.py**
- We begin by writing a class statement:

  class Card:
- Next we write the __**init**__ method. The **self** keyword refers to the object itself.
- **def __init__(self, n):**
-     **self._id = n**
- The __**init**__ method is called the class' **constructor** because it is used to construct new objects of the class

Building the Card Class

- Now we can write the **rank()** and **suit()** methods
- They translate the **_id** number into the rank and suit of a card

```
def suit(self):
    return self._id // 13
def rank(self):
    return self._id % 13
```

- This encoding ensures that all 13 cards of a single suit are placed together in consecutive order
- Now let's write a simple **__repr__** method

```
def __repr__(self):
    return 'Card #' + str(self._id)
```

Building the Card Class

- The **Card** class so far:

```
class Card:
    """ Instance variables: _id """
    def __init__(self, n):
        self._id = n

    def suit(self):
        return self._id // 13

    def rank(self):
        return self._id % 13

    def __repr__(self):
        return 'Card #' + str(self._id)
```

Building the Card Class (next)

- Suppose we created card #43: **c1 = Card(43)**
- If we go ahead and print out **c1**, we will get output like this:
- **Card #43**
- That's not very informative, so we'll have to fix it later
- We can write a function **new_deck()** that creates a list of 52 playing-card objects. This function is not part of the **Card** class itself. It is an example of use code of the **Card** class.

```
def new_deck():
    return [Card(i) for i in range(52)]
```

- An example call to this function:

```
deck = new_deck()
```

Building the Card Class

- Another improvement we can make is to add special methods that allow us to compare **Card** objects
- If we want to be able to sort **Card** objects, we must provide the **__lt__()** method, which tells us if one object is "less than" another:

```
def __lt__(self, other):
    return self._id < other._id
```

- **__eq__()** defines what it means for two **Card** objects to be "equal to" each other:

```
def __eq__(self, other):
    return self._id == other._id
```

Building the Card Class

- For example, consider the following objects:

  c1 = Card(1)

  c2 = Card(4)

- The expression **c1 < c2** would be **True**, but **c1 == c2** would be **False**
- Now that we can compare **Card** objects, we can sort them using the **sorted** function
  - **sorted** makes a copy of a list and then sorts the copy:

    cards_sorted = sorted(cards)

Building the Card Class

- The **Card** class defines an **application program interface** or API: a set of related methods that other programmers can use to build other software
- Also, we are applying the concept of encapsulation by gathering all the code that defines a **Card** object in one place
- On that topic, it can be useful to define **class variables**, values that pertain to a particular class but are not instance variables
- For our **Card** class it would be useful if we could print symbols representing the suits: ♣ ♦ ♥ ♠
- In Python we have access to many thousands of symbols
- We can access them by giving the correct numeric codes
- Let's add two class variables: **suit_sym** and **rank_sym** to **Card** class

Building the Card Class

- **suit_sym = {0: '\u2663', 1: '\u2666’,
  2: '\u2665', 3: '\u2660’}**
- If we were to print **suit_sym**, we would get this output:
- **{0: '♣', 1: '♦', 2: '♥', 3: '♠'}**
- The codes for various symbols can be found on the Internet by searching for “**Unicode characters**”
- Likewise, we can define a dictionary for all the ranks:
- **rank_sym = {0: '2', 1: '3', 2: '4', 3: '5', 4:
  '6', 5: '7', 6: '8', 7: '9', 8: '10', 9: 'J', 10:
  'Q', 11: 'K', 12: 'A'}**
- Our goal now is to be able to print a **Card** object in a form like “2♣”.
  Let’s see how to do that.

Building the Card Class

- We will change our definition of the **__repr__** method to this:

```
def __repr(self):
    return Card.rank_sym[self.rank()] +
              Card.suit_sym[self.suit()]
```

- Now, when we print a **Card** object, we will get output like 2♣, A♦, 8♠, J♠, etc.

Exceptions and Exception-handling

- What if another programmer using our class inadvertently gives a value outside the range 0 through 51 for **n** when constructing a **Card** object?
  - The __**init**__ method will accept the value, but it really shouldn't
- We can solve this problem by adding **exception handling** to our code
  - An ***exception*** is an unexpected event or error that has been detected
  - We say that the program has **raised** an exception
  - Let's have the __**init**__ method raise an exception if an invalid value is given for **n**

Exceptions and Exception-handling

```
def __init__(self, n):
    if n in range(0, 52):
        self._id = n
    else:
        raise Exception('Card number must be in the range 0-51.’)
```

- The new version of **__init__** verifies that the argument **n** is valid
- If not, it raises the exception and includes a diagnostic message of sorts

Exceptions and Exception-handling

- Consider a function now that a programmer might use to make new cards that catches any exception that might be thrown by the __**init**__ method:

```
def make_card(n):
    try:
        return Card(n)
    except Exception as e:
        print('Invalid card: ' + str(e))
        return None
```

- If we call **make_card(55)**, we get this output:

Invalid card: Card number must be in the range 0-51.

Exceptions and Exception-handling

- This concludes our development of the **Card** class
- See card.py for the completed **Card** class and use_card.py and use_card2.py for some tests
- Note: To run, drag use_card.py into PyCharm and run it. Be sure that card.py is in the same folder where use_card.py is located

Example: Acronym Generator (v1)

- Let’s explore a function that will create an acronym from the first letter of each “long” word in a list
- Define a “long” word to be any word with more than two letters
- After studying this first version, we will look at a second version that affords a little extra flexibility in creating acronyms

Example: acronym1.py

```
def acronym(phrase):
    result = ‘’
    words = phrase.split()
    for w in words:
        if len(w) >= 3:          # keep only long words
            result += w.upper()[0]
    return result
```

Trace: acronym() (version 1)

- Let’s trace the execution of this function for one example:

```
acronym('United States of America’)
def acronym(phrase):
    result = ‘’
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'

Trace: acronym() (version 1)

- Let's trace the execution of this function for one example:

```
acronym('United States of America')
def acronym(phrase):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	''

Trace: acronym() (version 1)

- Let's trace the execution of this function for one example:

```
acronym('United States of America')
def acronym(phrase):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	''
words	['United', 'States', 'of', 'America']

Trace: acronym() (version 1)

- Let's trace the execution of this function for one example:

```
acronym('United States of America')
def acronym(phrase):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	''
words	['United', 'States', 'of', 'America']
w	'United'

Trace: acronym() (version 1)

- Let's trace the execution of this function for one example:

```
acronym('United States of America')
def acronym(phrase):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) > 3:     # True
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	''
words	['United', 'States', 'of', 'America']
w	'United'

Trace: acronym() (version 1)

- Let’s trace the execution of this function for one example:

```
acronym('United States of America’)
def acronym(phrase):
    result = ‘’
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	'U'
words	['United', 'States', 'of', 'America']
w	'United'

Trace: acronym() (version 1)

- Let's trace the execution of this function for one example:

```
acronym('United States of America')
def acronym(phrase):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	'U'
words	['United', 'States', 'of', 'America']
w	'United'

Trace: acronym() (version 1)

- Let's trace the execution of this function for one example:

```
acronym('United States of America')
def acronym(phrase):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) > 3:     # True
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	'U'
words	['United', 'States', 'of', 'America']
w	'United'

Trace: acronym() (version 1)

- Let's trace the execution of this function for one example:

```
acronym('United States of America')
def acronym(phrase):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	'US'
words	['United', 'States', 'of', 'America']
w	'States'

Trace: acronym() (version 1)

- Let’s trace the execution of this function for one example:

```
acronym('United States of America’)
def acronym(phrase):
   result = ‘’
   words = phrase.split()
   for w in words:
      if len(w) > 3:
         result += w.upper()[0]
   return result
```

Variable	Value
phrase	'United States of America'
result	'US'
words	['United', 'States', 'of', 'America']
w	'of'

Trace: acronym() (version 1)

- Let’s trace the execution of this function for one example:

```
acronym('United States of America’)
def acronym(phrase):
   result = ‘’
   words = phrase.split()
   for w in words:
      if len(w) > 3:     # False
         result += w.upper()[0]
   return result
```

Variable	Value
phrase	'United States of America'
result	'US'
words	['United', 'States', 'of', 'America']
w	'of'

Trace: acronym() (version 1)

- Let’s trace the execution of this function for one example:

```
acronym('United States of America’)
def acronym(phrase):
    result = ‘’
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	'US'
words	['United', 'States', 'of', 'America']
w	'America'

Trace: acronym() (version 1)

- Let’s trace the execution of this function for one example:

```
acronym('United States of America’)
def acronym(phrase):
  result = ‘’
  words = phrase.split()
  for w in words:
    if len(w) > 3:     # True
      result += w.upper()[0]
  return result
```

Variable	Value
phrase	'United States of America'
result	'US'
words	['United', 'States', 'of', 'America']
w	'America'

Trace: acronym() (version 1)

- Let’s trace the execution of this function for one example:

```
acronym('United States of America’)
def acronym(phrase):
    result = ‘’
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	'USA'
words	['United', 'States', 'of', 'America']
w	'America'

Trace: acronym() (version 1)

- Let's trace the execution of this function for one example:

```
acronym('United States of America')
def acronym(phrase):
    result = ''
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
    return result
```

Variable	Value
phrase	'United States of America'
result	'USA'
words	['United', 'States', 'of', 'America']
w	'America'

Example: Acronym Generator (v2)

- Python allows function arguments to have **default values**
  - If the function is called without the argument, the argument gets its default value
  - Otherwise, the argument's value is given in the normal way
- We have seen a few examples of functions that have optional arguments
- A good example is the **round()** function, which takes two arguments: the value to round and an optional argument that indicates how many digits after the decimal point we want
  - If the second argument is not provided, the number of digits defaults to 0, e.g.,
    - round(4.56324) = 5
    - round(4.56324, 2) = 4.56

Example: Acronym Generator (v2)

The second version of **acronym** takes an optional argument, **include_shorts**, that tells the function to include the first letter of all words (including short words), but short words will **not** be capitalized if they are included

The first version of **acronym** simply discarded all short words

Example: acronym2.py

```
def acronym(phrase, include_shorts=False):
    result = ‘’
    words = phrase.split()
    for w in words:
        if len(w) > 3:
            result += w.upper()[0]
        elif include_shorts:
            result += w.lower()[0]
    return result
```

- By default, the optional argument is **False**, causing short words to be excluded
- When the optional argument is **True** and **w** is a short word, the first letter of the word in lowercase is concatenated to **result**

Example: acronym() (v2)

- Examples:

- **acronym('United States of America')** still returns **'USA'**
- **acronym('United States of America', True)** returns
- **'USoA'**

Optional Arguments

- As another example, suppose we want to make a revised version of the **bmi()** function from earlier in the course:

```
def bmi(weight, height):
    return (weight * 703) / (height ** 2)
```

- This version of **bmi()** assumes weight is given in pounds and height in total inches
- Suppose instead we want to give the programmer the option to use metric or standard (English) units
- We can add a third, optional argument, **units**, that defaults to metric if the programmer doesn't give a third argument
- Let's see the function on the next slide

Example: bmi_v4.py

```
def bmi(height, weight, units = 'metric’):
    if units == 'metric’:
        return weight / height**2
    elif units == 'standard’:
        return (weight * 703) / (height ** 2)
    else:
        return None
```

- Examples:

Examples:	Return Value:
bmi(100, 150, ‘standard’)	**10.545**
bmi(100, 150)	**0.015**
bmi(100, 150, ‘metric’)	**0.015**
bmi(100, 150, ‘unknown’)	**None**

Questions?