

# **CPSC 128 INTRODUCTION TO PROGRAMMING USING PYTHON**

Sinan Bulut, Dr.

# About me

**Physics (Condensed Matter)**

**Research: Trent, Augsburg, YRC**

**Scientific Programming**

**Supercomputers (HPC): ICHEC & GSC**

**Teaching: Trent, ICHEC**

# Course Philosophy

- 1. fundamentals of computer science & programming**
- 2. good programming:**
  - **easy to read (commenting & documentation)**
  - **easy to maintain & modify (version control, modular etc)**
  - **efficient (python libs and data structures, algorithms etc)**
  - **reliable (testing!)**

# Course Schedule

## 0. Course start-up.

### Part I: Procedural programming

#### 1. Introduction to computer science.

#### 2. SIPO (sequence, input, processing and output) programming.

#### 3. Selection control structures.

#### 4. Repetition control structures.

### Part II: Object-based programming

#### 5. Aggregate data types 1: Lists and strings.

#### 6. Functions.

#### 7. Aggregate data types 2: Dictionaries.

#### 8. Text files.

### Part III: Object-oriented programming (OOP)

#### 9. (OOP) 1: Encapsulation.

#### 10. Object-oriented design (OOD).

#### 11. (OOP) 2: Polymorphism and inheritance. Nov 24

#### ??12. Unified modeling language (UML).

#### Final Examination (Open “book”)

- May 2 - June 27: 8.5 weeks
- 1.5 modules / week
- Assignments: 65%
- Final: 35%

# What language?

Java

```
class myfirstjavaprogram
{
    public static void main(String args[])
    {
        System.out.println("Hi!");
    }
}
```

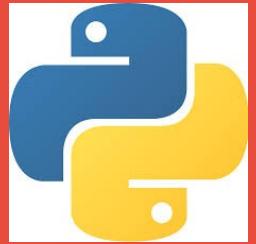
C++

```
#include <iostream>
int main()
{
    std::cout << "Hi!\n";
}
```

VS

Python

```
print "Hi!"
```



# Python

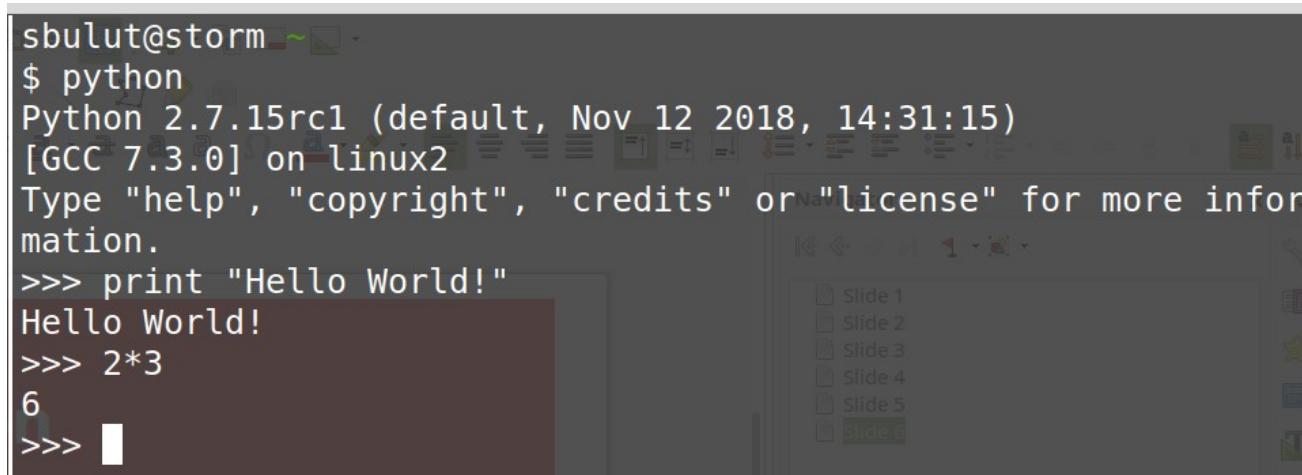
- **simple to learn & code**
- **general & wide application**
- **high level (not assembly)**
- **portable (runs on all OSs)**
- **extensive support libraries**

## Disadvantages:

- **interpreted & slow (like Matlab, Ruby etc) and high mem usage compared to C, Fortran, C++**

# Computing Setup: Python

REPL (Read, execute, print, loop) interface (also IDLE)

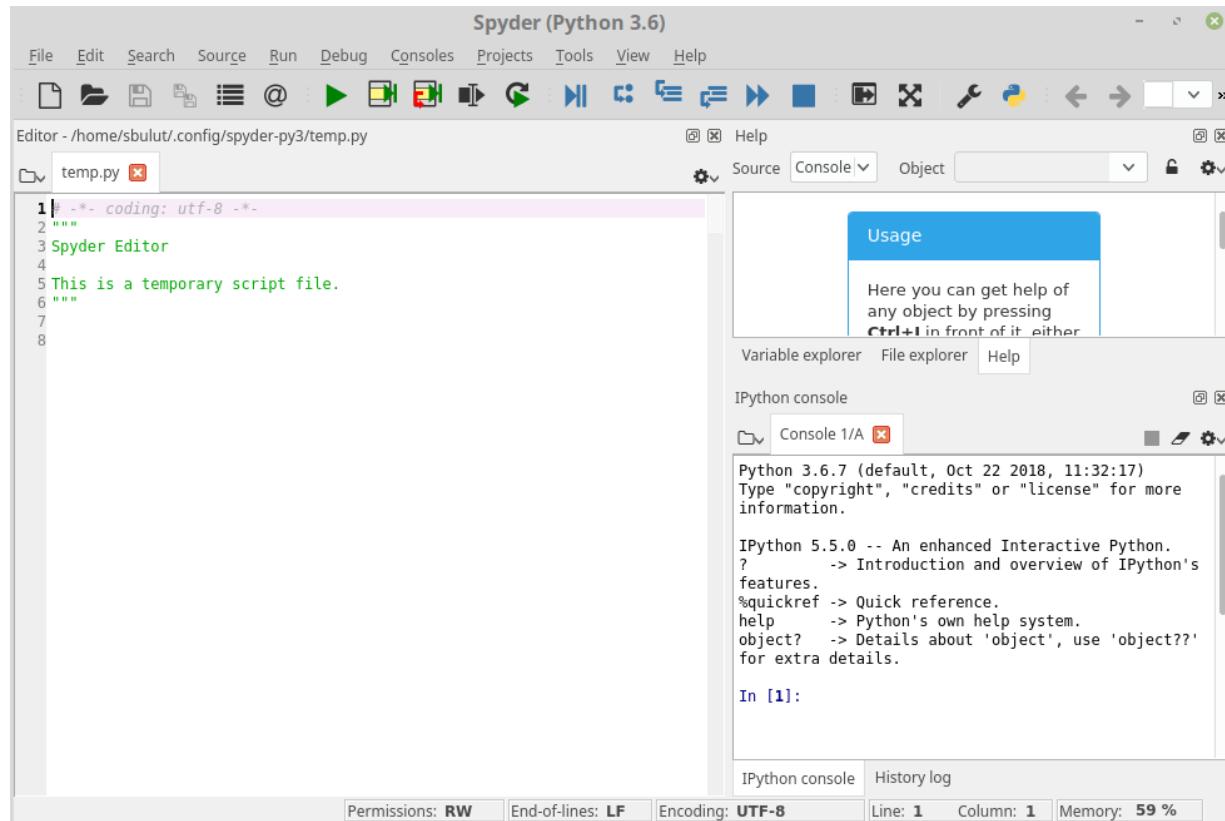


```
sbulut@storm:~$ python
Python 2.7.15rc1 (default, Nov 12 2018, 14:31:15)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World!"
Hello World!
>>> 2*3
6
>>>
```

- getting help: `help()`
- exiting: `exit()`
- Also: run a script from command line: `python hello.py`  
(in python3, parenthesis are required for `print`)

# Computing Setup: Python

Integrated Development environment (IDE): Spyder



# Computing Setup: Python

## Jupyter Notebook: (optional)

...allows you to create and share documents that contain live code, equations, visualizations and narrative text.

<https://jupyter.org/>



IP[y]: Notebook spectrogram Last Checkpoint: a few seconds ago (autosaved) IPython (Python 3)

File Edit View Insert Cell Kernel Help

Simple spectral analysis

An illustration of the [Discrete Fourier Transform](#) using windowing, to reveal the frequency content of a sound signal.

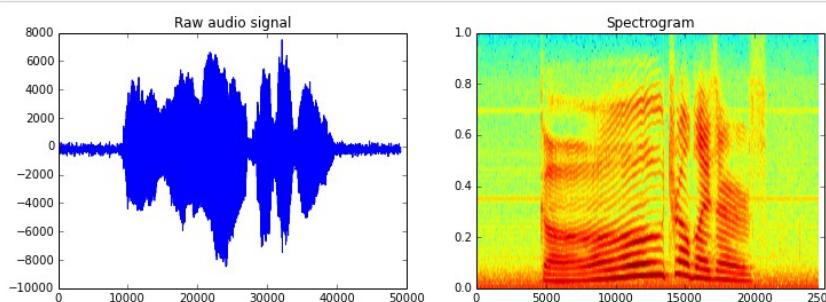
$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \quad k = 0, \dots, N - 1$$

We begin by loading a datafile using SciPy's audio file support:

```
In [1]: from scipy.io import wavfile  
rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin specgram routine:

```
In [2]: %matplotlib inline  
from matplotlib import pyplot as plt  
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))  
ax1.plot(x); ax1.set_title('Raw audio signal')  
ax2.specgram(x); ax2.set_title('Spectrogram');
```



# Computing Setup: Git

- **enables version control**
- **synchronize codes on multiple computers**
- **experiment with code without fear**
- **best practice for software development**

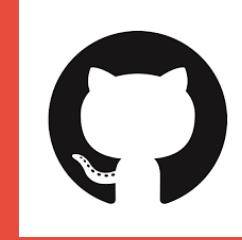
## Basic steps:

- “**git init .**” (**create a local repo: do only once**)
- “**git add somefile**”
- “**git commit -m “corrected typo, etc” somefile**

# Computing Setup: Git

```
File Edit View Search Terminal Help
sbultur@storm ~/Downloads/test_repo $ git init .
Initialized empty Git repository in /home/sbultur/Downloads/test_repo/
sbultur@storm ~/Downloads/test_repo $ git config user.email "Sinan81@earth.com"
sbultur@storm ~/Downloads/test_repo $ git config user.name "Sinan81"
sbultur@storm ~/Downloads/test_repo $ echo "Hello World" > file.txt
sbultur@storm ~/Downloads/test_repo $ git add file.txt
sbultur@storm ~/Downloads/test_repo $ git commit -m 'created a dummy file' file.txt
[master (root-commit) 9f612c5] created a dummy file
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt
sbultur@storm ~/Downloads/test_repo (master) $
```

# Computing Setup: Github



- **remote git repos are commonly used by devs to collaborate**
- **synchronize files from computer lab to home**
- **it is free!**
- **(will use ‘issues’ section for discussions)**

## Steps:

- **create account (use a simple passwd for now)**
- **create repo: “CPSC128”**
- **clone repo**
- **modify, commit changes, and push!**



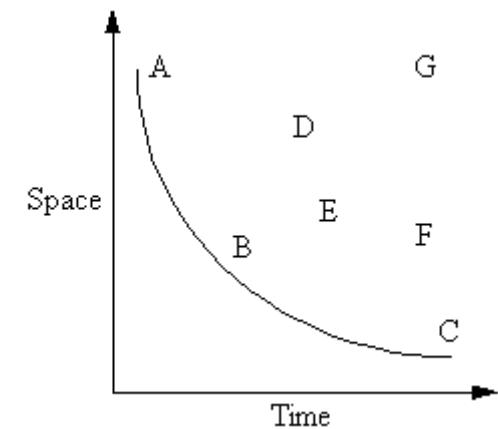
# Computing Setup: Github

```
File Edit View Search Terminal Help
sbulut@storm ~/Downloads GitHub, Inc. (US) | https://github.com/Sinan81/cpsc128
$ git clone https://github.com/Sinan81/cpsc128
Cloning into 'cpsc128'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 9 (delta 0), reused 6 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.
sbulut@storm ~/Downloads/cpsc128
$ cd cpsc128/
sbulut@storm ~/Downloads/cpsc128 (master)
$ git config user.email "Sinan81@earth.com"
sbulut@storm ~/Downloads/cpsc128 (master) Pull requests 0 Projects
$ git config user.name "Sinan81"
sbulut@storm ~/Downloads/cpsc128 (master)
$ echo "Hello World!" > README.md
sbulut@storm ~/Downloads/cpsc128 (master) provided.
$ git commit -m "a silly modification" README.md
[master f33217a] a silly modification
 1 file changed, 1 insertion(+), 2 deletions(-)
sbulut@storm ~/Downloads/cpsc128 (master)
$ git push
Username for 'https://github.com': Sinan81
Password for 'https://Sinan81@github.com':
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (1/1), done.
Writing objects: 100% (3/3), 255 bytes | 255.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/Sinan81/cpsc128
 5af4346..f33217a master -> master
sbulut@storm ~/Downloads/cpsc128 (master)
$
```

# Questions?

# Big Picture: Computer Science

- Computer Science is no more about computers than astronomy is about telescopes. (Edsger W. Dijkstra)
- Computation: To determine what can be computed and how it can best be computed
- The term “computer science” is a misnomer
- Computed?: the manipulation of data by a computational procedure (algorithm)
- Best?: algorithm that makes the most efficient use of our computational resources
- Computational resource?  
processor & memory



# Big Picture 2: Computers and Programs

- **Computer: a machine manipulating symbols**
- **Symbols: any discrete entity: letters, numbers...**
- **Programs: rules telling how to manipulate symbols**
  - Develop algorithm
  - Write computer program
  - Convert (compile) to binary (the language of computer)
  - Load to memory (cache) (done by OS)
  - Run/execute: program takes over the computer (done by OS)

# Examples of Computers



D-Wave (Vancouver)  
2000Qbit quantum computer



Raspberry Pi:  
starting at \$10



IBM-Q



IBM summit:  
~3Million processors  
10MW power  
<https://www.top500.org/>

[https://en.wikipedia.org/wiki/  
List\\_of\\_quantum\\_processors](https://en.wikipedia.org/wiki/List_of_quantum_processors)

# Big Picture: Programming

Computer Programming is a field that involves the methodology behind the programming, software abstraction, algorithms, data structures, design, testing, and maintenance of computer software. (Wikiversity)

Program: tells computer what to do

Catches:

1. expressing things in symbols
2. Comp's don't speak English
3. Comp languages can only refer to obj's in their world, not the natural world
4. Comp's can do simple manipulations, hence detailed instructions necessary

On the bright side:

1. World of comp's is a simple one: built on six key concepts
2. Prog languages contain very few words (33 keywords in Python)
3. A large set of techniques avail. for common situations, no need to start from scratch.

# The world of a computer: Six key op's

**The six operations consist of three elementary operations:**

**input**

**processing, and**

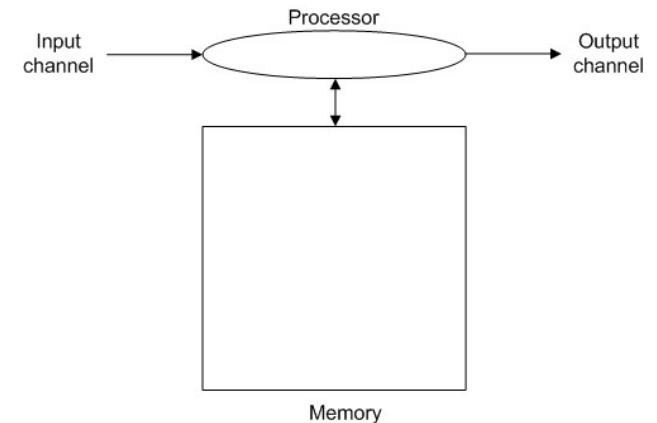
**output**

**and three control structures,**

**sequence**

**selection, and**

**repetition**



# **Input, processing, and output**

```
# input  
speed = input()  
duration = input()  
  
#processing  
distance = speed * duration  
  
#output  
print distance
```

# Sequential Execution

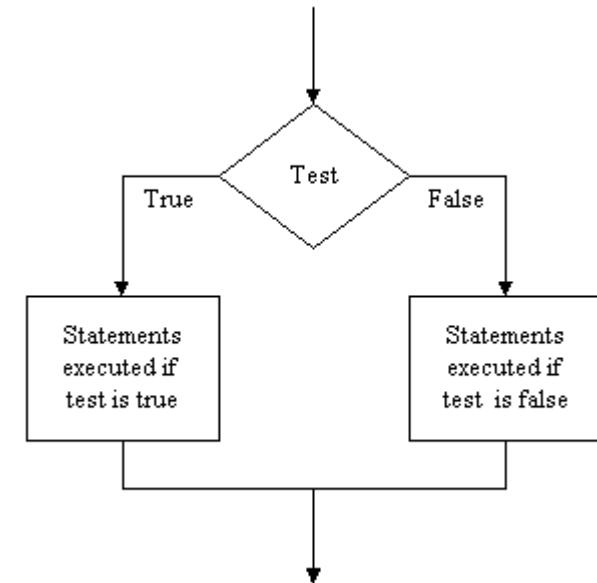
**By design, a program will execute sequentially, otherwise a program won't work:**

```
# This works as intended  
speed = input()  
duration = input()  
distance = duration * speed  
print distance
```

```
# This won't work  
speed = input()  
print distance  
distance = duration * speed  
duration = input()
```

# Selection

```
#  
if boxers_weight > 90:  
    print("Heavy weight")  
else:  
    print("Not heavy weight")
```



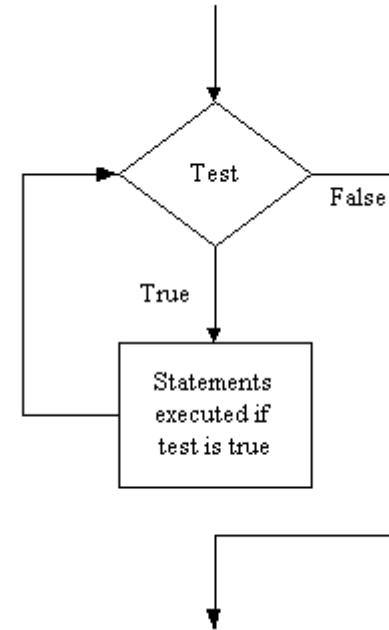
# Repetition

Some procedures require the ability to repeat a group of instructions:

- process the records in a file until the last line
- approximate  $f(x)$  until successive error is less than 0.001

**while** error > 0.001:

```
approx_new=... #newton-raphson  
error= abs(approx_old-approx_new)
```



# Exercise: play with Python shell

# First program: Fahrenheit to Celsius conversion

## The problem:

We need to convert a temperature measured on the Fahrenheit scale to its equivalent on the Celsius scale

Now we will work through the solution of this problem as follows:

- 1) Solving the problem by hand to make sure we know how to do it.
- 2) Consciously identifying our solution method, i.e. the solution algorithm.
- 3) Translating that algorithm into Python.
- 4) Entering and running the resulting program and verifying that it works.

# First program: Fahrenheit to Celsius conversion

## The problem:

We need to convert a temperature measured on the Fahrenheit scale to its equivalent on the Celsius scale

**Step1:** pick a temperature in Fahrenheit , and convert to Celsius by hand.

Using Wikipedia:  $C = (F - 32) \times 5/9$

Pick  $F=50$  degrees ->  $C=(50-32)*5/9=10$  degrees Celsius

# First program: Fahrenheit to Celsius conversion

## The problem:

We need to convert a temperature measured on the Fahrenheit scale to its equivalent on the Celsius scale

**Step1:** pick a temperature in Fahrenheit , and convert to Celsius by hand.

Using Wikipedia:  $C = (F - 32) \times 5/9$

Pick F=50 degrees ->  $C=(50-32)*5/9=10$  degrees Celsius

**Step2:** write it down as simple steps (i.e. algorithm) and think in terms of input, processing, output steps

1. get temp in F from user and save to a variable
2. calc temp in C
3. output C

# First program: Fahrenheit to Celsius conversion

## The problem:

We need to convert a temperature measured on the Fahrenheit scale to its equivalent on the Celsius scale

## Step3: Translate the algorithm into Python

Python 2.x      `temp_in_f = input()  
temp_in_c = (temp_in_f - 32) * 5 / 9  
print temp_in_c`

Python 3.x      `temp_in_f = eval(input())  
temp_in_c = (temp_in_f - 32)*5/9  
print(temp_in_c)`

## Step4: Run and test for a few F values

# First program: Fahrenheit to Celsius conversion

Can this code be improved ?

```
temp_in_f = eval(input())
temp_in_c = (temp_in_f - 32)*5/9
print(temp_in_c)
```

# First program: Fahrenheit to Celsius conversion

**Can this code be improved ?**

- Naming variables
- User interaction
- Documentation

```
temp_in_f = eval(input())
temp_in_c = (temp_in_f - 32)*5/9
print(temp_in_c)
```

# Naming variables

**We did a decent job in picking variable names:**

- **descriptive variable names (which enhance program readability)**
- **yet short, for convenience,**
- **all lowercase to adhere to Python convention**

What not to do (not descriptive!)

```
x = input()  
y = (x - 32) * 5 / 9  
print y
```

# User interface and user interaction

**What is wrong here?**

```
$ python3 f2c.py  
40  
4.444444444444445
```

# User interface and user interaction

**What is wrong here?**

```
$ python3 f2c.py  
40  
4.444444444444445
```

**Would user realize they are supposed to enter a value?**

**Would user think 4.44.. is output or think its just a random number or something?**

# User interface and user interaction

We need to provide instructions to the user, and explain output.

```
1  
2 print("This program converts temperatures from Fahrenheit to Celsius.")  
3 print("Enter a temperature in Fahrenheit (e.g. 10) and press Enter.")  
4 temp_in_f = eval(input("Temperature in Fahrenheit: "))  
5 #https://en.wikipedia.org/wiki/Fahrenheit  
6 temp_in_c = (temp_in_f - 32)*5/9  
7 print(temp_in_f, "degrees Fahrenheit =", temp_in_c, "degrees Celsius.")  
8
```

```
$ python3 f2c_v2.py  
This program converts temperatures from Fahrenheit to Celsius.  
Enter a temperature in Fahrenheit (e.g. 10) and press Enter.  
Temperature in Fahrenheit: 40  
40 degrees Fahrenheit = 4.444444444444445 degrees Celsius.)
```

# User interface and user interaction

```
1  
2 print("This program converts temperatures from Fahrenheit to Celsius.")  
3  
4 print("Enter a temperature in Fahrenheit (e.g. 10) and press Enter.")  
5  
6 temp_in_f = eval(input("Temperature in Fahrenheit: "))  
7  
8 #https://en.wikipedia.org/wiki/Fahrenheit  
9 temp_in_c = (temp_in_f - 32)*5/9  
10  
11 print(temp_in_f, "degrees Fahrenheit =", temp_in_c, "degrees Celsius.")  
12
```

**A few things to note here:**

- **variables & messages (string literals) can be combined in “print” statement using comma**
- **“input” statement can include a string literal as prompt**
- **we added blank lines to improve readability**

# Documentation

**Necessary step for producing complete program**

- **Meta comments: who, when, why created the code**
  - Use '#' (also possible to use docstring)
- **Inline comments: anything that puzzled you and you might not remember in 6 month from now**
- **Charts & user manual (for large projects)**

# Documentation: Example

```
1  
2 # f2c.py: converts a given temperature in Fahrenheit to Celsius  
3 # CPSC 128 Example program  
4 # S. Bulut, Spring 2018-19  
5  
6 print("This program converts temperatures from Fahrenheit to Celsius.")  
7  
8 print("Enter a temperature in Fahrenheit (e.g. 10) and press Enter.")  
9  
10 temp_in_f = eval(input("Temperature in Fahrenheit: "))  
11  
12 #https://en.wikipedia.org/wiki/Fahrenheit  
13 temp_in_c = (temp_in_f - 32)*5/9  
14  
15 print(temp_in_f, "degrees Fahrenheit =", temp_in_c, "degrees Celsius.")  
16
```

# Second program: c2f

**Now its your turn to write a program that converts from Celsius to Fahrenheit**

- follow the previously mentioned best practices
- save your code under -your- git repo
- once you are done do
  - git commit -m 'my first code' c2f.py
  - git push

# Last class recap & today

## Today:

- practice writing simple programs (SIPO)
- modulo op
- constants: why useful & best practice
- practice using `math` module
- int, float types & mixed arithmetic
- more ‘print’ tricks

# Third program: dhms2s

**Task: convert a time given in days, hours, minutes, and seconds to the equivalent number of seconds**

- do it by hand for a few input cases
- think what input, output should be what needs to be calculated
- translate into python
- check if code reproduces calc's done by-hand

# Third program: dhms2s

```
6 print("Enter your values now,")
7 days = eval(input("Enter days: "))
8 hours = eval(input("Enter hours: "))
9 minutes = eval(input("Enter minutes: "))
10 seconds = eval(input("Enter seconds: "))
11 tot_seconds = days*24*60*60 + hours*60*60 + minutes*60 + seconds
12 print("Total seconds is", tot_seconds)
13
```

Can we make this code faster?

# Third program: dhms2s

```
6 print("Enter your values now,")  
7 days = eval(input("Enter days: "))  
8 hours = eval(input("Enter hours: "))  
9 minutes = eval(input("Enter minutes: "))  
10 seconds = eval(input("Enter seconds: "))  
11 tot_seconds = days*24*60*60 + hours*60*60 + minutes*60 + seconds  
12 print("Total seconds is", tot_seconds)  
13
```

Can we make this code faster?

```
tot_seconds = ((days*24 + hours)*60 + minutes)*60 + seconds
```

# s2dhms

**Now lets write a program that does the inverse calculation:**

**take a large number of seconds and find the equivalent number of days, hours, minutes, and seconds**

- do it by hand (consider 200,000 seconds)
- determine: input, output, processing steps

# s2dhms

```
10 tot_seconds = eval(input("Enter the number of seconds: "))
11 days = tot_seconds // (24*60*60)
12 remainder = tot_seconds - days * (24*60*60)
13 hours = remainder // (60*60)
14 remainder = remainder - hours * (60*60)
15 minutes = remainder // 60
16 remainder = remainder - minutes*60
17 print(tot_seconds, "seconds is", days, "days,")
18 print(hours, "hours,", minutes, "minutes and")
19 print(remainder, " seconds.")
```

Let's improve this code!

# s2dhms

$$\text{days} = 200,000 / (24 * 60 * 60) = 2$$

$$\text{remainder} = 200,000 - 2 * (24 * 60 * 60) = 27,200$$

$$\text{hours} = 27,200 / (60 * 60) = 7$$

$$\text{remainder} = 27,200 - 7 * (60 * 60) = 2000$$

$$\text{minutes} = 2,000 / 60 = 33$$

$$\text{seconds} = 2,000 - 33 * 60 = 20$$

- the **input** is the number of seconds (200,000),
- the **processing** consists of several steps to find the number of days, hours, minutes and seconds, including intermediate steps to find the remainders along the way,
- the **output** is the number of days, hours, minutes and seconds.

# modulo: 5<sup>th</sup> arithmetic operator

- yields the remainder of a division
- denoted by ‘%’

example:  $7\%3 = \text{remainder of } 7/3 = 1$

- everyday arithmetic:  $-, +, /, *$
- remainder calc is needed in many computat'ns
- exist in most programming languages

# s2dhms: rewrite using modulo

```
days = tot_seconds / (24*60*60)
remainder = tot_seconds % (24*60*60)
hours = remainder / (60*60)
remainder = remainder % (60*60)
minutes = remainder / 60
remainder = remainder % 60
```

Can we do better?

# s2dhms: improve via symbolic constants

```
days = tot_seconds / (24*60*60)
remainder = tot_seconds % (24*60*60)
hours = remainder / (60*60)
remainder = remainder % (60*60)
minutes = remainder / 60
remainder = remainder % 60
```

We calculate the same thing more than once,  
hence there inefficiency.

Also, its not clear why are we multiplying certain numbers etc.

Let's avoid these using symbolic constants

# s2dhms: improve via symbolic constants

```
SECS_PER_DAY = 24 * 60 * 60
```

```
SECS_PER_HOUR = 60 * 60
```

```
SECS_PER_MINUTE = 60
```

```
days = tot_seconds // SECS_PER_DAY
remainder = tot_seconds % SECS_PER_DAY
hours = remainder // SECS_PER_HOUR
remainder = remainder % SECS_PER_HOUR
minutes = remainder // SECS_PER_MIN
remainder = remainder % SECS_PER_MIN
```

- Calculate once, use many times
- pick sym. constant names in a descriptive way to increase clarity
- Python convention: use capitals for constants: a cue to the reader

# s2dhms: putting it all together

```
1# s2dhms.py -- converts a time in seconds to
2# its equivalent in days, hours, minutes and seconds.
3#
4# CPSC 128 Demonstration Program
5#
6# S. Bulut, Spring 2018-19
7
8SECS_PER_DAY = 24 * 60 * 60
9SECS_PER_HOUR = 60 * 60
10SECS_PER_MINUTE = 60
11
12# Input.
13print("====")
14print("    Seconds to Days, Hours, Minutes and Seconds")
15print("-----")
16print
17print("This program converts a number of seconds to its")
18print("equivalent in days, hours, minutes and seconds.")
19tot_seconds = eval(input("Enter the number of seconds now (e.g. 116529): "))
20print
21
22# Processing.
23days = tot_seconds // SECS_PER_DAY
24remainder = tot_seconds % SECS_PER_DAY
25hours = remainder // SECS_PER_HOUR
26remainder = remainder % SECS_PER_HOUR
27minutes = remainder // SECS_PER_MINUTE
28remainder = remainder % SECS_PER_MINUTE
29
30# Output.
31print(tot_seconds, "seconds =", days, "days,", hours, "hours,")
32print(minutes, "minutes and", remainder, "seconds")
```

# s2dhms: putting it all together

```
1# s2dhms.py -- converts a time in seconds to
2# its equivalent in days, hours, minutes and seconds.
3#
4# CPSC 128 Demonstration Program
5#
6# S. Bulut, Spring 2018-19
7
8SECS_PER_DAY = 24 * 60 * 60
9SECS_PER_HOUR = 60 * 60
10SECS_PER_MINUTE = 60
11
12# Input.
13print("====")
14print("    Seconds to Days, Hours, Minutes and Seconds")
15print("-----")
16print("This program converts a number of seconds to its")
17print("equivalent in days, hours, minutes and seconds.")
18tot_seconds = eval(input("Enter the number of seconds now (e.g. 116529): "))
19
20print()
21
22# Processing.
23days = tot_seconds // SECS_PER_DAY
24remainder = tot_seconds % SECS_PER_DAY
25hours = remainder // SECS_PER_HOUR
26remainder = remainder % SECS_PER_HOUR
27minutes = remainder // SECS_PER_MINUTE
28remainder = remainder % SECS_PER_MINUTE
29
30# Output.
31print(tot_seconds, "seconds =", days, "days,", hours, "hours,",)
32print(minutes, "minutes and", remainder, "seconds")
```

blank lines

Line continuation

# Data types: int and float

- most common data types to store numbers
- **int** -> integers (positive & negative whole numbers without integers)
- **float** -> floating point values (numbers with decimals)
- “5” is not the same as “5.” or “5.0” in Python and other lang’s
- use **float** for measurements etc, use **int** for counts
- avoid mixing **int**’s and **float**’s (especially in python2)
  - although Python does its best to convert things to a more general type
  - i.e. int -> float , float -> complex
  - subtle problems can arise
  - conversion takes extra time
- Not Good:  $(-40 + 32.0) * 5 / 9$       **Good:**  $(-40.0 + 32.0) * 5.0 / 9.0$
- Casting: `int(3.14) -> 3`    `float(4) -> 4.0`

# Data types: int and float (Aside)

```
>>> type(1.5)
<class 'float'>
>>> type(4)
<class 'int'>
>>> int(1.5)
1
>>> float(4)
4.0
>>> import sys
>>> import math
>>> print("max integer: ",sys.maxsize,"2^{},math.log(sys.maxsize,2),
"} or 10^{}, math.log(sys.maxsize),\"}")
max integer: 9223372036854775807 2^{ 63.0 } or 10^{ 43.668272375276
55 }
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp
=308, min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, d
ig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1
)
>>> █
```

# use math module: calculate diameter of a tree

Scott used a tape measure to measure the circumferences of all the trees in a regeneration plot, but the database wants him to enter diameters instead of circumferences. Assuming the trees are roughly circular in cross-section we can calculate the diameter from the circumference, because the diameter is twice the radius and the radius is the circumference divided by  $2\pi$ . Write a program that will let him enter the circumference and will print the corresponding diameter. A sample run might look like:

# use math module: calculate diameter of a tree

```
8 import math
9 import math as m
10 from math import pi
11 #from math import *
12
13 circ = eval(input("Enter circumference:"))
14 print()
15 diameter = circ/math.pi
16 diameter2 = circ/m.pi
17 diameter3 = circ/pi
18
19 print("Diamter is ",diameter)
20 print("Diamter is : %1.2f" % diameter)
21 print("Diamter is : %1.2f" % diameter2)
22 print("Diamter is : %1.2f" % diameter3)
23
```

# SELECTION WITH IF (MAY14)

# Overview

- so far we've learned how to write basic simple programs (SIPO)
- it is often the case the processing of input depends on the situation (properties of input),
- we should be able to select what code to execute
- hence today we will learn if statement enabling selection in our codes

# **if: a simple example**

Check if a number is negative or positive, and accordingly output a message.

- SIPO statements are insufficient: processing depends on input
- Hence we need a selection statement: **if**

# if: a simple example

Check if a number is negative or positive, and accordingly output a message.

- SIPO statements are insufficient: processing depends on input
- Hence we need a selection statement: **if**

```
value = eval(input("Enter an integer (e.g, 23 or -118): "))
if value < 0:
    print(value, "is negative")
else:
    print(value, " is positive")
```

- evaluate **value < 0**
- if **True** execute first **print**
- if **False** skip first **print**, execute second **print**
- Either case, only one of **print** statements executed!
- Hence **if** selects what to execute based on a test

# if: a simple example

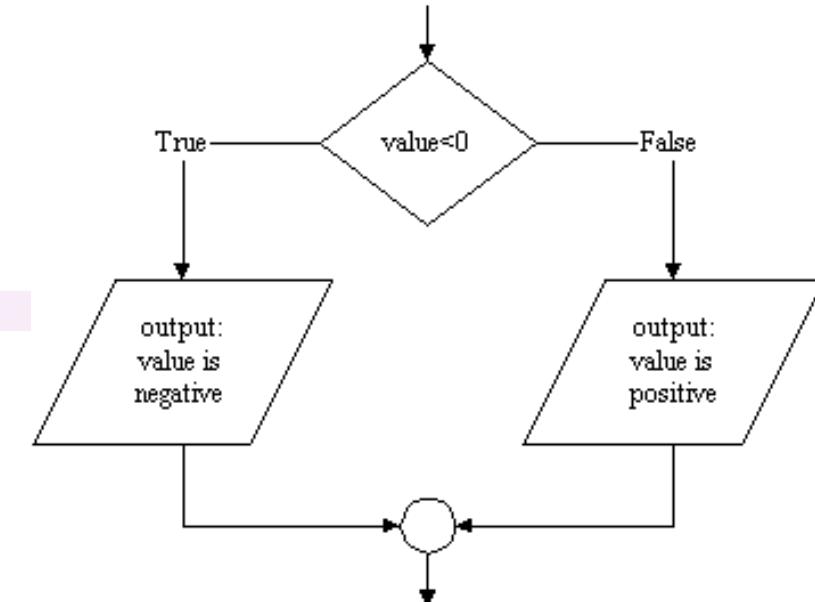
Check if a number is negative or positive, and accordingly output a message.

- SIPO statements are insufficient: processing depends on input
- Hence we need a selection statement: **if**

```
value = eval(input("Enter an integer (e.g., 23 or -118): "))
if value < 0:
    print(value, "is negative")
else:
    print(value, " is positive")
```

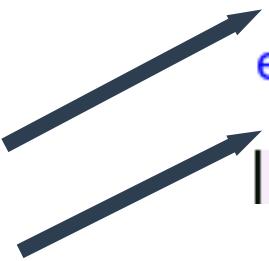
- evaluate **value < 0**
- if **True** execute first **print**
- if **False** skip first **print**, execute second **print**

- Either case, only one of **print** statements executed!
- Hence **if** selects what to execute based on a test



# if: a simple example

```
value = eval(input("Enter an integer (e.g, 23 or -118): "))
if value < 0:
    print(value, "is negative")
else:
    print(value, " is positive")
```



Note that the statements after the if and else statements are indented four spaces.

# An improvement

What is the output for  
input '0'?

```
value = eval(input("Enter an integer (e.g, 23 or -118): "))
if value < 0:
    print(value, "is negative")
else:
    print(value, " is positive")
```

# An improvement

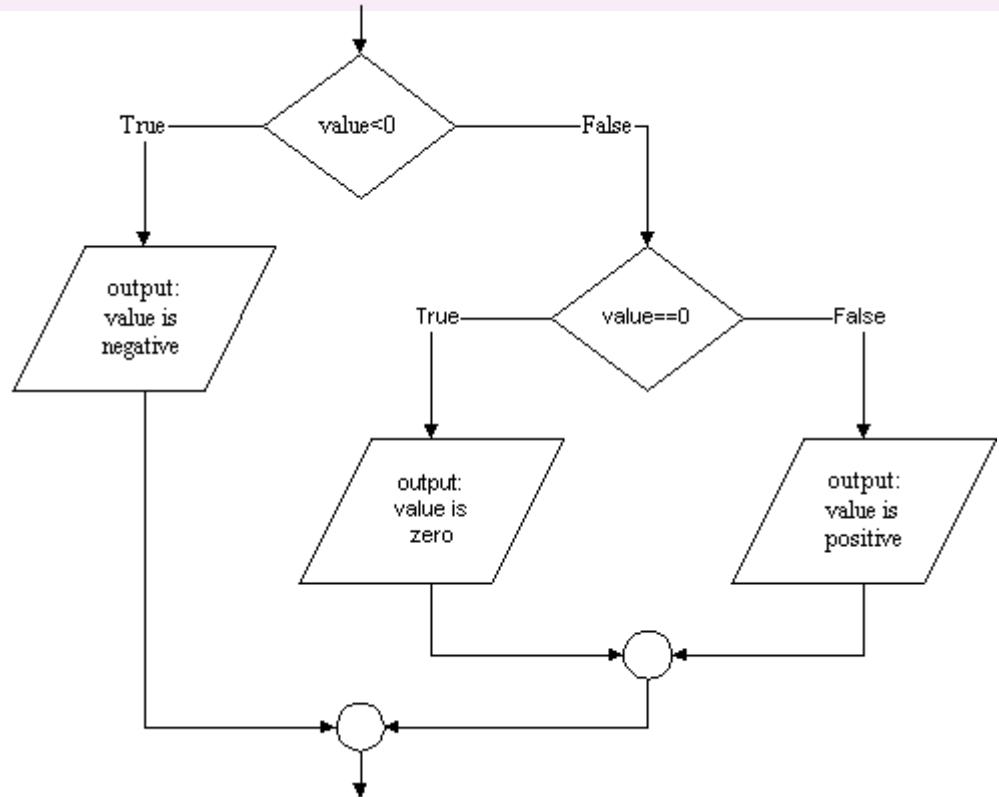
0 is not positive!

hence consider 3 cases:

1. value is negative
2. value is 0
3. value is positive

Consider the 3<sup>rd</sup> possibility via ->

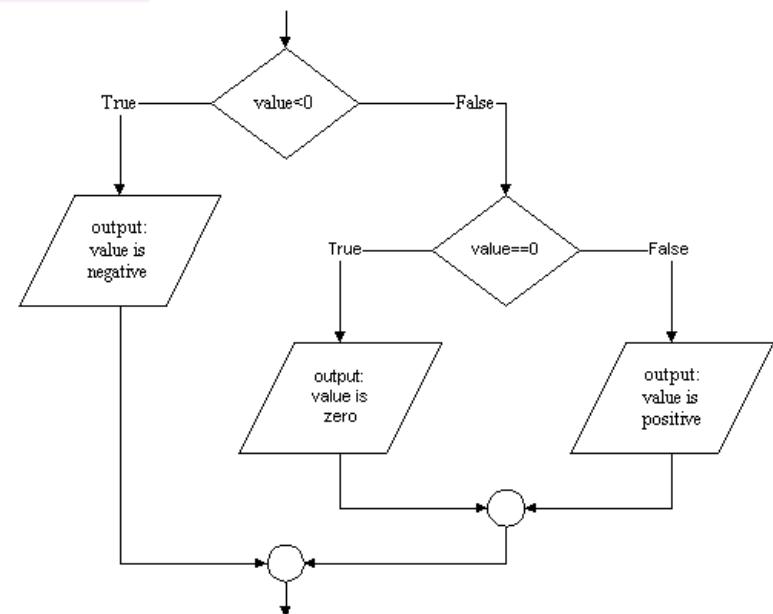
```
value = eval(input("Enter an integer (e.g, 23 or -118): "))
if value < 0:
    print(value, "is negative")
else:
    print(value, " is positive")
```



# An improvement: nested if statements

```
value = eval(input( "Enter an integer (e.g. 23 or -118): " ))
if value < 0:
    print(value, "is negative")
else:
    if value == 0:
        print(value, "is neither positive nor negative")
    else:
        print(value, "is positive")
```

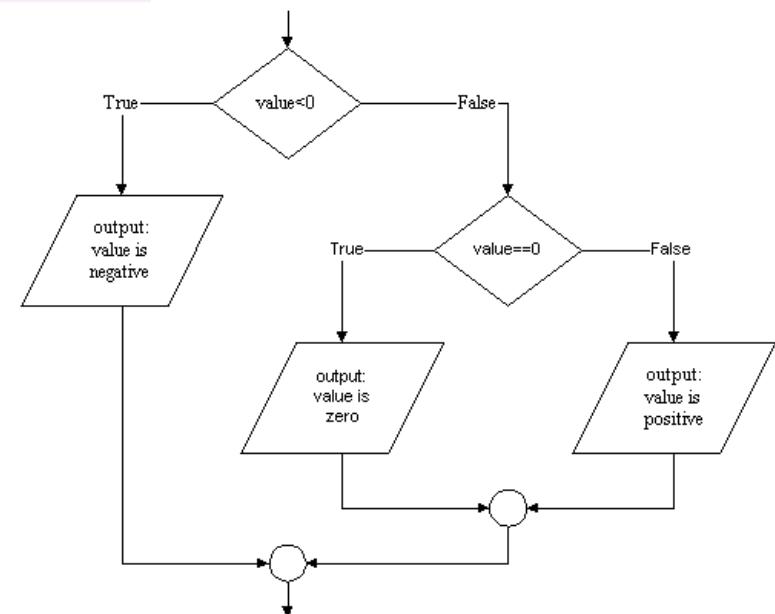
- Hence, we can nest 2 **if** statements
- Note the '==' operator
- Note the second level of indentation



# An improvement: nested if statements

```
value = eval(input( "Enter an integer (e.g. 23 or -118): " ))
if value < 0:
    print(value, "is negative")
else:
    if value == 0:
        print(value, "is neither positive nor negative")
    else:
        print(value, "is positive")
```

- Hence, we can nest 2 **if** statements
- Note the '==' operator
- Note the second level of indentation



# Multiway decision: **elif**

- Multiway decision (as opposed to two-way) is very common,  
so Python provides a special syntax: **elif**

```
value = eval(input( "Enter an integer (e.g, 23 or -118): " ))
if value < 0:
    print(value, "is negative")
elif value == 0:
    print(value, "is neither positive nor negative")
else:
    print(value, "is positive")
```

**elif** combines else & if

# if syntax

- Evaluates expressions one by one until one is found to be **True**
- Then executes corresponding statement
- Rest of **if** is skipped!!!
- if all expressions are false, executes else section & false statement

```
if test-expression1:  
    true-statement1  
[elif test-expression2:  
    true-statement2]  
...  
[else:  
    false-statement]
```

# if syntax

- Evaluates expressions one by one until one is found to be **True**
- Then executes corresponding statement
- Rest of **if** is skipped!!!
- if all expressions are false, executes else section & false statement

```
value = eval(input( "Enter an integer (e.g. 23 or -118): "| ))  
  
if value < -100:  
    print(value, "is very negative")  
elif value < 0:  
    print(value, "is negative")  
elif value == 0:  
    print(value, "is neither positive nor negative")  
else:  
    print(value, "is positive")
```

What is the output  
for inputs  
“-20” and “-101”

# Relational Expressions

operator	description	try in Python shell!
<	less than	
<=	less than or equal to	
==	equal	
!=	not equal	
>=	greater than or equal to	
>	greater than	

# Relational Expressions

Python also provides logical operators that can be combined with relational operators:

```
age >= 20 and age < 30
```

```
age >= 20 or income < 18.45
```

```
In [54]: not True  
Out[54]: False
```

```
In [55]: not False  
Out[55]: True
```

x	y	x and y	x or y
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

x	not x
0	1
1	0

# Example: A single program to convert F to C or C to F

```
print("This program converts temperatures from Fahrenheit to Celsius,")
print("or from Celsius to Fahrenheit.")
print("Choose")
print("1 to convert Fahrenheit to Celsius")
print("2 to convert Celsius to Fahrenheit")

choice = eval(input( "Your choice? " ))|  
  
if choice == 1:
    print("This program converts temperatures from Fahrenheit to Celsius.")
    temp_in_f = eval(input( "Enter a temperature in Fahrenheit (e.g. 10) and press Enter: "))
    temp_in_c = (temp_in_f - 32.0)*5.0/9.0
    print(temp_in_f, " degrees Fahrenheit = ", temp_in_c, " degrees Celsius.")

elif choice == 2:
    print("This program converts temperatures from Celsius to Fahrenheit .")
    temp_in_c= input( "Enter a temperature in Celsius (e.g. 10) and press Enter: ")
    temp_in_f = temp_in_c *9.0/5.0 + 32.0
    print(temp_in_c, " degrees Celsius = ", temp_in_f, " degrees Fahrenheit.")

else:
    print("Error: Your choice not recognized!")
```

# Case Study: Converting numerical grades to letter grades

Write a program  
that acceptst numerical grade  
and outputs letter grade

Numerical Grade	Letter Grade
95-100	A+
86-94	A
80-85	A-
75-79	B+
70-74	B
65-69	B-
62-64	C+
58-61	C
55-57	C-
50-54	D
0-50	F

# Case Study: Converting numerical grades to letter grades

```
# Version 1.  
print " ... " # Instructions to user  
grade = eval(input(" ... "))  
if grade >= 95 and grade <= 100:  
    print("A+")  
if grade >= 86 and grade <= 94:  
    print("A")  
if grade >= 80 and grade <= 85:  
    print("A-")  
# ...  
# and so on down to  
# ...  
if grade >= 50 and grade <= 54:  
    print("D")  
if grade >= 0 and grade <= 50:  
    print("F")  
if grade < 0 or grade > 100:  
    print("Error: The grade must be between 0 and 100.")
```

# Case Study: Converting numerical grades to letter grades (version 2)

```
# Version 2.  
print(" ... ") # Instructions to user  
grade = eval(input(" ... "))  
if grade >= 95 and grade <= 100:  
    print("A+")  
elif grade >= 86 and grade <= 94:  
    print("A")  
elif grade >= 80 and grade <= 85:  
    print("A-")  
# ...  
# and so on down to  
# ...  
elif grade >= 50 and grade <= 54:  
    print("D")  
elif grade >= 0 and grade <= 50:  
    print("F")  
else:  
    print("Error: The grade must be between 0 and 100.")
```

# Case Study: Converting numerical grades to letter grades (version 3)

```
# Version 3.  
print(" ... ") # Instructions to user  
grade = eval(input("..."))  
if grade > 100 or grade < 0:  
    print("Error: The grade must be between 0 and 100.")  
elif grade >= 95:  
    print("A+")  
elif grade >= 86:  
    print("A")  
elif grade >= 80:  
    print("A-")  
# ...  
# and so on down to  
# ...  
elif grade >= 50:  
    print("D")  
elif grade >= 0:  
    print("F")
```

# Case Study: Testing

- Let's test this code for a few grades:

99, 13, 27

- What values do you think we should choose for testing?

```
# Version 3.  
print(" ... ") # Instructions to  
grade = eval(input("..."))  
if grade > 100 or grade < 0:  
    print("Error: The grade must  
elif grade >= 95:  
    print("A+")  
elif grade >= 86:  
    print("A")  
elif grade >= 80:  
    print("A-")  
# ...  
# and so on down to  
# ...  
elif grade >= 50:  
    print("D")  
elif grade >= 0:  
    print("F")
```

# Case Study: Testing

- Let's test this code for a few grades:

99, 13, 27

- Programs usually break due to uncommon values and unexpected inputs
  - \* 75.3 (float input)
  - \* edge cases: 0, 100
  - \* out of range inputs: -1, 101 etc
  - \* garbage input: can program handle it?

```
# Version 3.  
print(" ... ") # Instructions to  
grade = eval(input("..."))  
if grade > 100 or grade < 0:  
    print("Error: The grade must  
elif grade >= 95:  
    print("A+")  
elif grade >= 86:  
    print("A")  
elif grade >= 80:  
    print("A-")  
# ...  
# and so on down to  
# ...  
elif grade >= 50:  
    print("D")  
elif grade >= 0:  
    print("F")
```

# Case Study: Performance Analysis

Number of comparison & logical operations			
	Minimum	Average	Maximum
Version 1	12 if/elifs @ 3 = 36	12 if/elifs @ 3 = 36	12 if/elifs @ 3 = 36
Version 2	3	(12 if/elifs @ 3 )/2 = 18	12 if/elifs @ 3 = 36
Version 3	3	((1 if @ 3) + (11 if/elifs @ 1 ))/2 = 7	((1 if @ 3) + (11 if/ellifs @ 1 )) = 14

- further improvement: check for the most likely grades first  
does performance gain justify a less readable code?
- above performance analysis involves overestimation  
as Python ‘short circuits’ test expressions when it can

# Summary: selection using `if`

```
if test-expression:  
    true-statement
```

```
if test-expression:  
    true-statement  
else:  
    false-statement
```

```
if test-expression:  
    true-statement  
elif test-expression:  
    true-statement  
...  
else:  
    false-statement
```

- three ways of using `if`
- **else** is usually used for catching errors or invalid values!

# Programming Exercise: Gazinta (as in "2 Gazinta 8, but 3 doesn't")

sample runs:

```
Wondering if one number "goes into" another?  
Give me the numbers (big one first) and I'll tell you.  
First number = 8  
Second number = 2  
2 does go into 8 exactly.
```

```
Wondering if one number "goes into" another?  
Give me the numbers (big one first) and I'll tell you.  
First number = 8  
Second number = 3  
3 does NOT go into 8 exactly.
```

```
Wondering if one number "goes into" another?  
Give me the numbers (big one first) and I'll tell you.  
First number = 8  
Second number = 0  
Sorry division by 0 is undefined!
```

Problem:

Write a program that accepts two numbers, and outputs a message telling the user whether the second number divides exactly into the first or not.

# Programming Exercise: Gazinta (as in "2 Gazinta 8, but 3 doesn't")

```
1# gazinta.py -- this code takes in two numbers, and determines if one divides the first one exactly.
2# S. Bulut May 2019
3
4
5print("""Wondering if one number "goes into" another?
6Give me the numbers (big one first) and I'll tell you.""")
7
8a = eval(input("First number = "))
9
10b = eval(input("Second number = "))
11
12if b == 0:
13    print("Sorry division by 0 is undefined!")
14elif a%b == 0:
15    print(b, " does go into", a, "exactly")
16else:
17    print(b, " does NOT go into", a, "exactly")
```

# Programming exercise: Pythagorean theorem

Problem:

Write a program that accepts three integer values, and outputs a message stating whether they could be the sides of a right-angled triangle or not.

```
Enter the lengths of the sides of your triangle and I will tell you  
if it has a right angle or not.  
Length of first side = 3  
Length of second side = 4  
Length of third side = 5  
That is a right-angled triangle.
```

```
Enter the lengths of the sides of your triangle and I will tell you  
if it has a right angle or not.  
Length of first side = 3  
Length of second side = 5  
Length of third side = 4  
That is a right-angled triangle.
```

```
Enter the lengths of the sides of your triangle and I will tell you  
if it has a right angle or not.  
Length of first side = 3  
Length of second side = 4  
Length of third side = 6  
That is NOT a right-angled triangle.
```

```
Enter the lengths of the sides of your triangle and I will tell you  
if it has a right angle or not.  
Length of first side = 4  
Length of second side = 4  
Length of third side = 4  
That is NOT a right-angled triangle.
```

# Programming exercise: Pythagorean theorem (Solution)

```
# pythagoras.py
# S. Bulut 2019
# T. Topper 2015

print('''
=====
Right-angle tirangle tester
-----')

Enter the lengths of the sides of your triangle and I will let you know
if it has a right angle or not
''')

a = eval(input("Length of the first side ="))
b = eval(input("Length of the second side ="))
c = eval(input("Length of the third side ="))

if a**2 + b**2 == c**2 or a**2 + c**2 == b**2 or b**2 + c**2 == a**2:
    print("That is a right angled triangle")
else:
    print("That is NOT a right angled triangle.")
```

# Programming exercise: Pythagorean theorem (Solution)

```
# pythagoras_b.py
# S. Bulut 2019
# T. Topper 2015

print("""
=====
Right-angle tirangle tester
-----

Enter the lengths of the sides of your triangle and I will let you know
if it has a right angle or not
""")

a = eval(input("Length of the first side ="))
b = eval(input("Length of the second side ="))
c = eval(input("Length of the third side ="))

# find hypoteneuse
hypo = a
if b > hypo:
    hypo = b

if c > hypo:
    hypo = c

if a**2 + b**2 + c**2 == 2*hypo**2
    print("That is a right angled triangle")
else:
    print("That is NOT a right angled triangle.")
```

# Programming Exercise: Utility Bills

Problem:

A utility company takes readings each month and charges its customers according to the table shown below.

Write a program to calculate a customer's bill amount given the number of kilo-watt hours used.

kWh	Rate
Under 500	\$20.00
500 to 1000	\$20.00 + \$0.03 per kWh over 500
Over 1000	\$35.00 + 0.02 per kWh above 1000

```
=====
```

Bill calculator

```
-----
```

Enter kilowatt-hours used: 300

Bill amount = \$20.00

```
=====
```

Bill calculator

```
-----
```

Enter kilowatt-hours used: 600

Bill amount = \$23.00

```
=====
```

Bill calculator

```
-----
```

Enter kilowatt-hours used: 1200

Bill amount = \$39.00

# Programming Exercise: Utility Bills (Solution)

```
# utilitybill.py: calculates utility bill for a give consumption in kWh
# S. Bulut 2019
# T. Topper 2015

print('
=====
Bill Calculator
-----
''')

kwh = eval(input("Enter kWh used: "))
if kwh < 0:
    print('\nRecheck input value!')
    print("It was negative and meters\ndon't run backwards")
else:
    if kwh <= 500.0:
        amount = 20.0
    elif kwh <= 1000.0:
        amount = 20.0 + 0.03*(kwh-500.)
    else:
        amount = 35.0 + 0.02*(kwh-1000.)
```

# REPETITION CONTROL STRUCTURES

## (May 15)

- so far learned 5 out of 6 key concepts:  
**input, processing, output, sequence, & selection**
- repetition will enable expressing any  
**computational algorithm**
- many algorithms involve performing an action  
many times until a condition is met
- python provides: while and for
- by end of today, will learn how to use these  
**statements in your program**

# The idea: go back up and repeat some code

- until now our programs were strictly sequential
- “if” enables skipping of some code but program still flows from top to bottom
- many situations require being able to go back up and repeat certain sections of code



```
print "Instructions to user..."  
grade = input("Enter a numerical grade between 0 and 100: ")  
  
if grade>100 or grade<0:  
    print "Error: The grade must be between 0 and 100."  
elif grade>=95:  
    print "A+"  
elif grade>=86:  
    print "A"  
...  
...
```

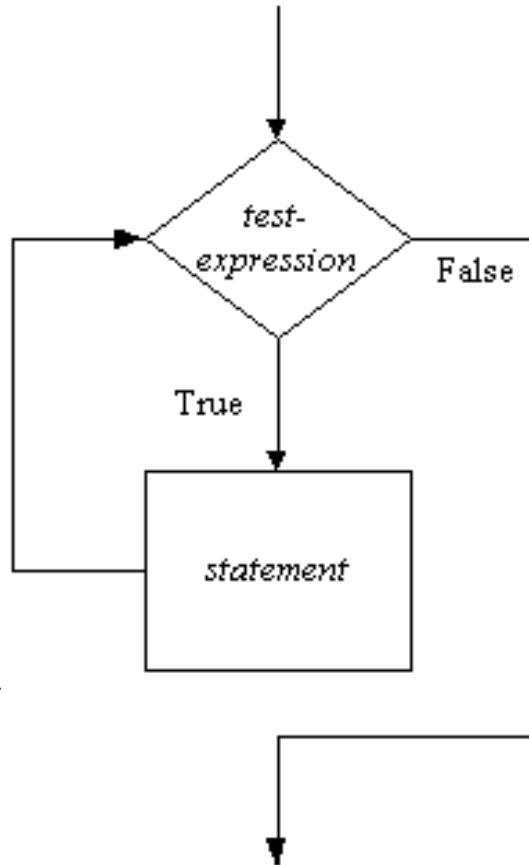
# while loop:

## Syntax

```
while test-expression:  
    statement
```

usage scenario:

repeat code as many times as necessary



# Example: Five Hi!

```
# example: five hi
counter = 5
while counter >0:
    print("Hi!")
    counter=counter-1
```

....  
Hi!  
Hi!  
Hi!  
Hi!  
Hi!

```
# example: five hi
counter = 5
while counter >0:
    print("Hi!")
    counter -= 1
```

augmented assignment

# Beware of infinite loops!

```
# infinite loop
while True:
    print("Help! I'm stuck in a loop!")
```

```
Help! I'm stuck in a loop!
```

# Input validation

```
validgrade = False #initialize
while not validgrade:
    print("please enter a valid grade between 0 and 100")
    grade = eval(input("Enter grade"))
    validgrade = grade >= 0 and grade <=100
```

```
.....
please enter a valid grade between 0 and 100

Enter grade
-1
please enter a valid grade between 0 and 100

Enter grade
101
please enter a valid grade between 0 and 100

Enter grade
27
```

# Input validation

```
validgrade = False #initialize
while not validgrade:
    print("please enter a valid grade between 0 and 100")
    grade = eval(input("Enter grade"))
    validgrade = grade >= 0 and grade <=100

# input validation
while True:
    print("please enter a valid grade between 0 and 100")
    grade = eval(input("Enter grade\n"))
    validgrade = grade >= 0 and grade <=100
    if validgrade:
        break
```

# Repeating a program

```
# repeat a program
# here is a structure to enable user run a code again
# without having to restart the program
again='y'
while again=='y' or again=='Y':
    #
    # put the code you want to repeat here
    #
    again = input("Do it again (y/n)? ")
print("bye now")
```

Note: we're not doing “eval(input....”

# Repeating a program

## Example: f2c

```
# f2c_while.py: converts a given temperature in Fahrenheit to Celsius
# CPSC 128 Example program
# S. Bulut, Spring 2018-19

again='y'
while again=='y' or again=='Y':

    print("This program converts temperatures from Fahrenheit to Celsius.")
    print("Enter a temperature in Fahrenheit (e.g. 10) and press Enter.")
    temp_in_f = eval(input("Temperature in Fahrenheit: "))
    temp_in_c = (temp_in_f - 32)*5/9
    print(temp_in_f, "degrees Fahrenheit =", temp_in_c, "degrees Celsius.")

    again = input("Do it again (y/n)? ")

print("bye now")
```

In [39]: runfile('/home/sbulut/github/cpsc128/code/python3/f2c\_wl  
code/python3')

This program converts temperatures from Fahrenheit to Celsius.  
Enter a temperature in Fahrenheit (e.g. 10) and press Enter.

Temperature in Fahrenheit: 32  
32 degrees Fahrenheit = 0.0 degrees Celsius.

Do it again (y/n)? y  
Enter a temperature in Fahrenheit (e.g. 10) and press Enter.

Temperature in Fahrenheit: -40  
-40 degrees Fahrenheit = -40.0 degrees Celsius.

Do it again (y/n)? n  
bye now

In [40]:

# Example: Find average

Write a program that can be used to find the average of a set of numbers

# Example: Sentinel value controlled loop

Re-write `find_average.py` with a sentinel value controlled loop

# Programming practice: Smallest and Largest values

Write a program that reads a sequence of integer values typed by the user and then displays the smallest and largest values entered. Use a sentinel value to stop the reading of the values, and allow the user to specify the value of the sentinel.

# Programming practice: Smallest and Largest values

```
# sentinel.py
# Find the smallest and largest values in a sequence entered by the user.
#
# Tim Topper
# Fall 2008

print "This program will display the largest and smallest values you enter."
print
print "You have to choose a special number to indicate the end of your input."
sentinel = input("What number will indicate that you have entered all your values? ")
print

print "Begin entering your values now."
print "Press Enter after each one, and type your special number when you are done."
num = input("> ")
smallest = num
largest = num
while num != sentinel:
    if num < smallest:
        smallest = num
    elif num > largest:
        largest = num
    num = input("> ")

if smallest == largest == sentinel:
    print "You only entered the special number you chose."
else:
    print "The smallest value in your list is: %d" % (smallest)
    print "The largest value in your list is: %d" % (largest)
```

# Problem: The guessing game

There is a guessing game children are fond of in which one player chooses a number between 1 and 100 and the other player must guess it. The only feedback given the guesser is whether their most recent guess was too high or too low. The game ends when the number has been guessed.

Write a program that 'plays' this game by picking a number and then accepting guesses about the number's value. After each guess it should inform the user if the guess was too high or too low. The game should end when the number is guessed, at which point the computer should display the number of guesses taken.

**Hint:** in order to get a random number between 0 and 100, do =>

```
In [59]: import random  
In [60]: print(random.randint(1,100))  
53  
In [61]: print(random.randint(1,100))  
79
```

# Playing computer: A puzzle: What output does this produce?

Step through code and keep track of values (on white board or paper).

```
# playing computer v1
x = 0
while x < 5:
    y = 0
    while y < 3:
        print("*")
        y=y+1
    print()
    x = x+1
```

# Playing computer: (version 2)

## A puzzle: What output does this produce?

```
# playing computer v2
x = 0
while x < 5:
    y = 0
    while y < x:
        print("*")
        y=y+1
    print()
    x = x+1
```

# Nested loops

- we just saw example of nested loops

The syntax is:

```
# nested while
while test1:
    [statement]
    while test2:
        statement
```

The key point:

inner loop does **all** of its iterations for **each** of the outer loop iterations!!!

# Python's other repetition structure: for loops

The syntax is:

```
for item in sequence:  
    statement
```

"for each item in sequence execute the following statement[s]"

Examples of sequences (iterables):

- string            'hello'
- list              [1, 2, 3]
- tuple             (1, 2, 3)
- dictionary       {'a': 97, 'b': 98}
- set               {"apple", "banana", "cherry"}

```
# for loop examples  
for c in 'This is a vertical text!':  
    print(c)  
  
for x in [1, 2, 3]:  
    print(x)  
  
for x in range(3):  
    print(x)
```

# Five hi! revisited

```
# example: five hi
counter = 5
while counter >0:
    print("Hi!")
    counter=counter-1|
```

```
# five hi! for loop
for x in [1, 2, 3, 4, 5]:
    print("Hi!")
```

- Not all repetitions can be represented using **for** loop
- **while** is more general and flexible,

# Nested for loops

## Exercise: draw a square

Write a program that reads in the size of a square and a character and then prints out a hollow square of that size out of the specified character and blanks. Your program should work for squares of all sizes from 1 to 20 inclusive. Here is a sample run of the program,

```
What size square? 5
What character? *
*****
*   *
*   *
*   *
*****
*****
```

Think about pseudocode first!

Hint: **print(c, end="")** prints a character without moving on to the next line  
(note default end='\n' &  
print(c) => used default end character.

Hint: you might want to use **range()** to generate a sequence.

# Nested for loops

## Exercise: draw a square (solution v1)

```
9 print("This program will draw a hallow square on the screen")
10 print()
11
12 size = eval(input("How large a square would you like?"))
13 c = input("Which character should I use to draw it?")
14
15 if size < 1:
16     print("Sorry, can't handle negative sizes or null size.")
17 elif size == 1:
18     print('c')
19 else:
20     # draw top row of the square
21     for i in range(size):
22         print(c, end='')
23     print()
24
25 #    # draw the middle rows of the square
26 for i in range(size-2):
27     print(c, end='')
28     for j in range(size-2):
29         print(' ',end='')
30     print(c)
31
32 #    # draw the bottom row of the square
33 for i in range(size):
34     print(c, end='')
35 print()
```

draw\_a\_square\_v1.py

# Nested for loops

## Exercise: draw a square (solution v2)

```
1 # draw_a_square_v2.py
2 # Draw a hallow square on screen. The size and character to use are chosen
3 # by the user.
4 #
5 # CPSC 128 example code
6 # S. Bulut
7 # May 2019
8
9 print("This program will draw a hallow square on the screen")
10 print()
11
12 size = eval(input("How large a square would you like?"))
13 c = input("Which character should I use to draw it?")
14
15 if size < 1:
16     print("Sorry, can't handle negative sizes or null size.")
17 elif size == 1:
18     print('c')
19 else:
20     for row in range(size):
21         for col in range(size):
22             if row==0 or row==size-1 or col==0 or col==size-1:
23                 print(c, end='')
24             else:
25                 print(' ', end='')
26     print()
```

# Nested for loops

## Exercise: draw a square (solution v3)

```
1# draw_a_square_v3.py
2# Draw a hallow square on screen. The size and character to use are chosen
3# by the user.
4#
5# CPSC 128 example code
6# S. Bulut
7# May 2019
8
9print("This program will draw a hallow square on the screen")
10print()
11
12size = eval(input("How large a square would you like?"))
13c = input("Which character should I use to draw it?")
14
15if size < 1:
16    print("Sorry, can't handle negative sizes or null size.")
17elif size == 1:
18    print('c')
19else:
20    print(size*c+'\n'+(size-2)*(c+(size-2)*' '+c+'\n')+size*c+'\n')
21
```

# Summary: repetition structures in Python

```
while test-expression:  
    statement  
for item in sequence:  
    statement
```

# Applications of loops: Brute force method

*When in doubt, use brute force. ~ Ken Thompson, Bell Labs*

- We are not quick in doing processing and we are unreliable in doing repetitive operations.
- However, computers are very quick and reliable in doing repetitive calcs
- Hence, using loops, we can find brute force solutions for many computational problems
- Let's consider examples of simulations and number crunching

# Number crunching: Programming exercise: Factors

Write a program that inputs a positive whole number, and displays all the number's factors, i.e. all the numbers that divide into it exactly. For instance if the number 12 is input, the values 1, 2, 3, 4, 6, and 12 should be output.

# Number crunching: Programming exercise: Factors

Write a program that inputs a positive whole number, and displays all the number's factors, i.e. all the numbers that divide into it exactly. For instance if the number 12 is input, the values 1, 2, 3, 4, 6, and 12 should be output.

```
 7 print("This program will display all factors of
 8 num = eval(input("Enter an integer number"))
 9 print()
10
11 print("%d's factors are: " % num)
12 # alternative print statement with sep=' ' trick
13 # print(num,"'s factors are:", sep=' ')
14 divisor=1
15 while divisor <= num:
16     if num%divisor == 0:
17         print(divisor, num/divisor)
18     divisor=divisor+1
19
```

```
11 print("%d's factors are: " % num)
12 # alternative print statement with sep=' '
13 # print(num,"'s factors are:", sep=' ')
14 for divisor in range(1,num+1):
15     if num%divisor == 0:
16         print(divisor, num/divisor)
17
```

# Simulation

In a simulation problem we write a program that simulates a real world process, relying on our ability to generate random numbers to take account of the uncertainty of real-world processes like whether

- newborns are girls or boys,
- how long it will take to unload an airplane,
- the chances a vehicle will turn left or right or go straight.

If we do this enough times we can make a reliable estimate of the probability of a real-world event occurring

# Problem: What are the chances: four children, all girls?

Write a simulation program that estimates the percentage of 4-child families in which all the children are girls by simulating the formation of 1000 four-child families. Assume that the chances of a newborn being a girl or a boy are equal. A sample run of the program might look like this:

```
In 1,000 simulated four-child families  
approximately 7% were made up of four daughters.
```

Hint: use ‘random.randint()’ to flip a coin to see if a birth event will yield a boy or a girl

# Problem: What are the chances: four children, all girls?

```
1# allgirls.py
2
3# simulate 1000 4-child families
4# count how many of them have all-daughters
5import random
6
7allgirls = 0
8for family in range(1000):
9
10    daughters = 0
11    for birth in range(4):
12        # assign 0 -> a boy, 1-> a girl
13        if random.randint(0,1) == 1:
14            daughters = daughters + 1
15
16    if daughters == 4:
17        allgirls = allgirls + 1
18
19
20print("Percentage of all girl families is: %1.2f" % (allgirls/1000.*100) )
21
```

Re-do: using while loop!

# Part II Object based programming

- Aggregate data types 1: Lists and strings.
- Functions.
- Aggregate data types 2: Dictionaries.
- Text files.

# Object based programming

- all the values that we've worked with are obj's
- We did things like 'x = a + 10' using built-in operators,
- We did polymorphism/operator overloading

x = a + 10      vs s='hello' + 'world'

9%5              vs print("%d apples" % 9)

In python everything is an object:

An object is a collection of attributes and methods/functions that it knows how to apply to itself

# Object based programming

Example: name='alice' is a string object  
its attribute is a being sequence of the characters 'a' 'l' 'i' 'c' 'e'  
its methods:

```
>>> dir(name)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__formatter_field_name_split__', '__formatter_parser__', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>> 
```

- methods with double underscore '\_\_' correspond to methods invoked by an operator such as '+' corresponding to '\_\_add\_\_'
- others invoked via syntax:

*object\_name.method\_name(parameters)*

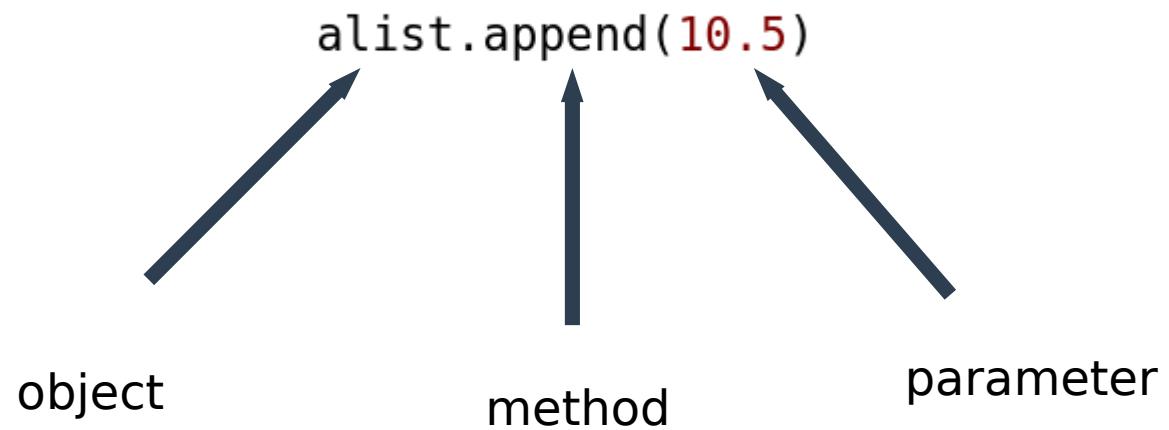
# Object based programming

```
In [31]: name='alice'
```

```
In [32]: name.upper()  
Out[32]: 'ALICE'
```

```
In [33]: name.center(10)  
Out[33]: ' alice '
```

```
In [34]: name.split()  
Out[34]: ['alice']
```



# Sequence Types

Do the following examples:

## Strings

```
name = "Joe"  
"m" in name  
"o" not in name  
min(name)  
max(name)  
name = name + " " + "Sixpack"  
print(name)  
name[0]  
name[2]  
name[-1]  
name[-2]  
name[2:7]  
len(name)  
name*3
```

## Lists

```
nums = [1, 2, 3, 4, 5]  
3 in nums  
9 in nums  
min(nums)  
max(nums)  
nums = nums + [9, 10]  
print(nums)  
nums[0]  
nums[2]  
nums[-1]  
nums[-2]  
nums[2:5]  
len(nums)  
nums[2] = 11  
print(nums)  
nums[2:6] = [20, 30]
```

Note: Lists are mutable while strings aren't!

# What can you do with a list?

## - **create a list**

mylist = range(10)

mylist = [19, 'a', 3.14, 'Tea', False]

## - **access an item**

1<sup>st</sup> item: mylist[0] is 19

2<sup>nd</sup> item: mylist[1] is 'a'

last item: mylist[-1] is False

## - **access a slice**

mylist[1:4] is ['a', 3.14, 'Tea']

note range is [inclusive:exclusive]

a slice is always a list:

mylist[1:2] = ['a'] not 'a'

# What can you do with a list?

## - add items to a list

mylist.append(2) -> [19, 'a', 3.14, 'Tea', False, 2]

mylist.insert(1, 'hi') -> [19, 'hi', 'a', 3.14, 'Tea', False, 2]

mylist = mylist + ['bye'] -> [19, 'hi', 'a', 3.14, 'Tea', False, 2, 'bye']

## - remove items

by position: del( mylist[2] )

by value: mylist.remove( 'a' )

from the end: mylist.pop()

# What can you do with a list?

## - get information

test to see if an item is in the list: if 'Bye' in myList:

len(myList) gives number of items in the list

myList.index('Tea') returns the position of item 'Tea'

myList.count('bye') counts the number of 'bye's in the list

## - miscellaneous

myList.sort()

myList.reverse()

myList.extend()

2\*myList

# What can you do with a string?

## - create a string

```
name=input('enter your name')  
name='John Sixpack'
```

## - access item/s

name[0] is the first character in the string: 'J'  
name[7] is the 8<sup>th</sup> character in the string: 'x'  
name[1:7] is a slice: 'ohn S'

## - add to a string

strings are immutable: so create a new string and assign to same variable  
name = name + ' hello' -> 'John Sixpack hello'  
name = name[0:4] + ' hello' + name[4:12] -> John hello Sixpack

# What can you do with a string?

## - getting information

if 'a' in name:

len(name)

name.find('ola') returns position where the first occurrence of 'ola'

## - misc.

name.count('p')

name[2].isalpha()

':'.join(['abc', 'def', 'ghi'])

"867-395-0892".split('-')

" John Sixpack ".strip()

5\*'+'

# Example: Playlist shuffle

Problem: There are N songs in a given playlist. Shuffle this playlist.

example playlist: shuffled playlist: [4,3,11,..., N, ..., 1, 7]

Hint: use list data type and random.randint()

pseudocode?

# Example: Playlist shuffle

Problem: There are N songs in a given playlist. Shuffle this playlist.

example playlist: shuffled playlist: [4,3,11,..., N, ..., 1, 7]

```
Initialize playlist (like getting a blank sheet of paper)  
Until you have six numbers in the playlist
```

```
    Roll the die
```

```
    If the value of the die is not in the playlist  
        Add it to the playlist
```

# Example: Playlist shuffle

Problem: There are N songs in a given playlist. Shuffle this playlist.

example playlist: shuffled playlist: [4,3,11,..., N, ..., 1, 7]

```
1 # playlist_shuffle.py
2
3
4 import random
5
6 nsongs = eval(input("How many songs are in the playlist?"))
7
8 playlist=[]
9 while len(playlist) < nsongs:
10     song = random.randint(1,nsongs)
11     if song not in playlist:
12         playlist.append(song)
13
14 print(playlist)
```

# Dice Odds

Imagine that before we go on vacation to Vegas we want to quickly figure out some of the relevant odds. Let's start with the frequency of various outcomes when rolling a pair of dice. We could do this by rolling a pair of fair dice many times and recording (counting!) how many times each outcome occurs. There are 11 possible outcomes because we can observe anywhere from 2 to 12 spots on the pair of dice. This means we need to record how many twos occur, how many threes occur, how many fours occur, and so on up to how many twelves occur.

Hint: use lists in storing outcome and/or occurrences.

Sample output for 1000 dice rolls.

Outcome	Occurrences
2	29
3	66
4	71
5	100
6	137
7	173
8	132
9	122
10	78
11	59
12	33

# Dice Odds (solution)

```
1 import random
2 #import time as tm
3 #start = tm.time()
4
5 ROLLS = 1000
6
7 # Initialize counters to 0.
8 counters = [0] * 11
9
10# Roll dice many times and record frequency of outcomes.
11for roll in range(ROLLS):
12    outcome = random.randint(1, 6) + random.randint(1, 6)
13    # Now increment the appropriate counter.
14    counters[outcome-2] = counters[outcome-2] + 1
15
16# Display results.
17print(" =====")
18print(" Outcome | Occurrences")
19print(" -----+-----")
20for posn in range(11):
21    print("%7d |%8d" % (posn+2, counters[posn]))
22print(" -----")
23
24#end = tm.time()
25#print("Calculation completed in ",end-start," seconds")
```

**compare with the version without using a list!**

dice\_roll\_v1.py

# Representing Playing Cards

- in the previous example we used a list to store an array of numbers
- lists can also be used to store information about real world objects with multi component attributes:
- Example: playing cards

# Representing Playing Cards

0 Ace	Clubs	13 Ace	Diamonds	26 Ace	Hearts	39 Ace	Spades
1 Two	Clubs	14 Two	Diamonds	27 Two	Hearts	40 Two	Spades
2 Three	Clubs	15 Three	Diamonds	28 Three	Hearts	41 Three	Spades
3 Four	Clubs	16 Four	Diamonds	29 Four	Hearts	42 Four	Spades
4 Five	Clubs	17 Five	Diamonds	30 Five	Hearts	43 Five	Spades
5 Six	Clubs	18 Six	Diamonds	31 Six	Hearts	44 Six	Spades
6 Seven	Clubs	19 Seven	Diamonds	32 Seven	Hearts	45 Seven	Spades
7 Eight	Clubs	20 Eight	Diamonds	33 Eight	Hearts	46 Eight	Spades
8 Nine	Clubs	21 Nine	Diamonds	34 Nine	Hearts	47 Nine	Spades
9 Ten	Clubs	22 Ten	Diamonds	35 Ten	Hearts	48 Ten	Spades
10 Jack	Clubs	23 Jack	Diamonds	36 Jack	Hearts	49 Jack	Spades
11 Queen	Clubs	24 Queen	Diamonds	37 Queen	Hearts	50 Queen	Spades
12 King	Clubs	25 King	Diamonds	38 King	Hearts	51 King	Spades

-52 unique cards

-13 face values

-four suits

- if we can agree on ordering,  
each card can be represented by a number between 0-51
- so use the above standard contract-bridge ordering

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ..., 47, 48, 49, 50, 51]
```

hence **deck = list(range(52))**

# Representing Playing Cards: Suit from card number

Solution #1

```
if cardnum < 13 :  
    suit = 'Clubs'  
elif cardnum < 26 :  
    suit = 'Diamonds'  
elif cardnum < 39 :  
    suit = 'Hearts'  
else :  
    suit = 'Spades'
```

Solution #2

```
suit = cardnum / 13  
if suit == 0 :  
    suit = 'Clubs'  
elif suit == 2 :  
    suit = 'Diamonds'  
elif suit == 3 :  
    suit = 'Hearts'  
else :  
    suit = 'Spades'
```

# Representing Playing Cards: Face value from card number

Solution #1: straightforward conversion from card-id to face value

```
if cardnum == 0 or cardnum == 13 or cardnum == 26 or cardnum == 39 :  
    face_value = 'Ace'  
elif cardnum == 1 or cardnum == 14 or cardnum == 27 or cardnum == 40 :  
    face_value = 'Two'  
...
```

Can we improve this using lists?

# Representing Playing Cards: Face value from card number

**Solution #1:** straightforward conversion from card-id to face value

```
if cardnum == 0 or cardnum == 13 or cardnum == 26 or cardnum == 39 :  
    face_value = 'Ace'  
elif cardnum == 1 or cardnum == 14 or cardnum == 27 or cardnum == 40 :  
    face_value = 'Two'  
...  
...
```

**Solution #2:** improve sol#1 using list membership tests

```
if cardnum in [0, 13, 26, 39]:  
    face_value = 'Ace'  
elif cardnum in [1, 14, 27, 40]:  
    face_value = 'Two'  
elif cardnum in [2, 15, 28, 41]:  
    face value = 'Three'  
...  
...
```

# Representing Playing Cards: Face value from card number

```
face_value = cardnum % 13
if face_value == 0 :
    face_value = 'Ace'
elif face_value == 2 :
    face_value = 'Two'
elif face_value == 3 :
    face_value = 'Three'
elif face_value == 4 :
    face_value = 'Four'
elif face_value == 5 :
    face_value = 'Five'
elif face_value == 6 :
    face_value = 'Six'
elif face_value == 7 :
    face_value = 'Seven'
elif face_value == 8 :
    face_value = 'Eight'
elif face_value == 9 :
    face_value = 'Nine'
elif face_value == 10 :
    face_value = 'Ten'
elif face_value == 11 :
    face_value = 'Jack'
elif face_value == 12 :
    face_value = 'Queen'
else :
    face_value = 'King'
```

**Solution #3:** consider the remainder when card-index divided by 13

# Representing Playing Cards: Face value from card number

```
face_value = cardnum % 13
if face_value == 0 :
    face_value = 'Ace'
elif face_value == 2 :
    face_value = 'Two'
elif face_value == 3 :
    face_value = 'Three'
elif face_value == 4 :
    face_value = 'Four'
elif face_value == 5 :
    face_value = 'Five'
elif face_value == 6 :
    face_value = 'Six'
elif face_value == 7 :
    face_value = 'Seven'
elif face_value == 8 :
    face_value = 'Eight'
elif face_value == 9 :
    face_value = 'Nine'
elif face_value == 10 :
    face_value = 'Ten'
elif face_value == 11 :
    face_value = 'Jack'
elif face_value == 12 :
    face_value = 'Queen'
else :
    face_value = 'King'
```

<-- **Solution #3:** consider the remainder when card-index divided by 13

**Solution #4:** improve sol#3 by using list look-up

```
FACE_VALUES = ['Ace', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', \
                'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King']
face_value = cardnum % 13
print('The face value of card number', cardnum, 'is', FACE_VALUES[face_value])
```

Conc:

- using lists can make code more readable and easy to understand
- possibly the ones with lists will perform better than the case where there is a cascade of if statements.

# Representing Playing Cards: Dealing a Hand

Problem: How can we deal a hand of cards from our deck of cards?

```
import random

# Create the deck of cards.
deck = range(52)

# Shuffle the deck of cards
for swaps in range(104):
    posn1 = random.randint(0, 51)
    posn2 = random.randint(0, 51)
    # Swap the cards at posn1 and posn2
    (deck[posn1], deck[posn2]) = (deck[posn2], deck[posn1])

# Create the empty hand.
hand = []

# Deal 5 cards from the deck into the hand.
for card in range( 0, 5 ):
    hand.append( deck.pop() )
```

**Solution1:** shuffle the deck and then either:

- (i) use **.pop()** to select one random item at a time or
- (ii) use **list slicing** to get as many random cards as needed.

# Representing Playing Cards: Dealing a Hand

Problem: How can we deal a hand of cards from our deck of cards?

**Solution2:** randomly select cards from deck and add to the **hand** list

```
import random
deck = range(52)
hand = []
for card in range(5) :
    # Choose the card to deal.
    posn = random.randint(0, len(deck) - 1)
    # Append the number at that position to the hand.
    hand.append(deck[posn])
    # Delete that card from the deck.
    del(deck[posn])
```

**Question** Imagine a list with 1 Billion items,  
which solution method would you use?

# Representing Playing Cards: Dealing a Hand: Putting it together

```
1# deal_a_hand.py
2# This program deals a hand of 5 cards at random.
3# CPSC128 Example code
4# S. Bulut 2019, T. Topper 2015
5
6import random
7
8# Define handy string constants.
9FACE_VALUES = ['Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
10                  'Seven', 'Eight', 'Nine', 'Ten', 'Jack',
11                  'Queen', 'King']
12SUTITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
13
14# Create deck of cards.
15deck = list(range(52))
16
17# Create empty hand.
18hand = []
19
20# Deal 5 cards into hand.
21for deal in range(5):
22    posn = random.randint(0, len(deck) - 1)
23    hand.append(deck[posn])
24    del(deck[posn])
25
26# Display the cards in the hand.
27for card in hand:
28    print(FACE_VALUES[card % 13], 'of', SUTITS[card // 13])
```

# Programming Practice

## Poker hands: is it a “flush”?

**Task:** write code to detect if our hand is a flush, i.e. that all the cards in the hand are from the same suit.

since face values are irrelevant,

lets create a second list (lets call it **suit\_list**) corresponding to the suit type

For simplicity represent suit type in numbers:

0 -> club

1 -> diamond

2 -> heart

3 -> spade

**a hand:** [47, 36, 20, 40, 14]    **corresponding suit list :** [3, 2, 1, 3, 2]

Nine of Spades

Jack of Hearts

Eight of Diamonds

Two of Spades

Two of Diamonds

an example of flush woud be:

suit\_list: [2, 2, 2, 2, 2]

# Programming Practice

## Poker hands: is it a “flush”?

**Task:** write code to detect if our hand is a flush, i.e. that all the cards in the hand are from the same suit.

since face values are irrelevant,

lets create a second list (lets call it **suit\_list**) corresponding to the suit type

```
suit_list = []
for card in hand:
    suit_list.append(card//13)
```

# Programming Practice

## Poker hands: is it a “flush”? (Sol’n 1)

```
28 if suit_list[1] == suit_list[0] and suit_list[2] == suit_list[1] and \
29     suit_list[3] == suit_list[2] and suit_list[4] == suit_list[3]:
30     print("That's a flush!")
31 else:
32     print("Sorry no flush here.")
```

What if there is more than 5 cards in ‘hand’? Let’s find a more general solution

# Programming Practice

## Poker hands: is it a “flush”? (Sol’n 1)

```
34## is it a flush? Sol_1b
35flush = True # Start by assuming it is a flush.
36
37# The first test compares the second card to the first card,
38# so the initial previous card is the first card in the hand.
39prev_card = suit_list[0]
40for card in suit_list[1:]:
41    # Note: To loop starting with the second card we use a slice of the list,
42    # that begins in the second position, i.e. position #1.
43    if card != prev_card: # If card's suit not the same as previous one's.
44        flush = False # It's not a flush.
45    prev_card = card # Update previous card: This card will be the previous
46                      # card next time around.
47if flush:
48    print("That's a flush!")
49else:
50    print("Sorry, no flush here.")
51
```

further improvement: can we get rid of ‘prev\_card’?

# Programming Practice

## Poker hands: is it a “flush”? (Sol’n 1)

```
53## Is it a flush? sol_1c
54flush = True
55for posn in range(1,len(suit_list)-1):
56    if suit_list[posn] != suit_list[posn-1]:
57        flush = False
58        break
59
60if flush:
61    print("That's a flush!")
62else:
63    print("Sorry, no flush here.")
```

# Programming Practice

## Poker hands: is it a “flush”? (Sol’n 2)

Another approach would be to count how many cards there are of each suit. If the count is ever the same as the length of the hand then it's a flush.

```
66## Is it a flush? sol2
67flush = False
68for suit in [0,1,2,3]:
69    count = 0
70    for card_suit in suit_list:
71        if card_suit == suit:
72            count = count + 1
73    if count == len(suit_list):
74        flush = True
75    break
76
77if flush:
78    print("That's a flush!")
79else:
80    print("Sorry, no flush here.")
81
```

# Programming Practice

## Poker hands: is it a “flush”? (Sol’n 3)

Yet another approach takes advantage of some of the built-in list methods. For example the count method counts how many times a value occurs in a list.

```
83 if suit_list.count(suit_list[0]) == len(suit_list):  
84     print("That's a flush!")  
85 else:  
86     print("Sorry, no flush here.")  
--
```

# Programming Practice

## Poker hands: is it a “flush”? (Sol’n 4)

Another approach that leverages Python's built-in list methods:

When sorted, the first and last item in a suit\_list should be the same in a flush scenario:

[2, 1, 3, 0, 2] -> sorted -> [0, 1, 2, 2, 3]  
above, first and last items aren't the same  
[2, 2, 2, 2, 2] -> sorted -> [2, 2, 2, 2 ,2]  
first and last items are the same! its a flush!

```
89 suit_list.sort()
90 if suit_list[0] == suit_list[len(suit_list) - 1]:
91     print("That's a flush!")
92 else:
93     print("Sorry, no flush here.")
```

# Poker hands: summary

With solutions 3 and 4 being so short why waste time on long solutions like 1 and 2?

- not all languages provide the rich set of list methods that Python does
- solutions 1 and 2 showed techniques that are useful in a wide variety of problems:
  1. comparing elements of a list pairwise
  2. looping from part way through a list to the end
  3. looping through a list by index position
  4. using flag variables
  5. using counters

**There's (almost) always more than one way to solve it!**

# Poker Hands: alternative representation

```
deck = [
    ['A', 'C'], ['2', 'C'], ['3', 'C'], ['4', 'C'], ['5', 'C'], ['6', 'C'], ['7', 'C'],
    ['8', 'C'], ['9', 'C'], ['10', 'C'], ['J', 'C'], ['Q', 'C'], ['K', 'C'],
    ['A', 'D'], ['2', 'D'], ['3', 'D'], ['4', 'D'], ['5', 'D'], ['6', 'D'], ['7', 'D'],
    ['8', 'D'], ['9', 'D'], ['10', 'D'], ['J', 'D'], ['Q', 'D'], ['K', 'D'],
    ['A', 'H'], ['2', 'H'], ['3', 'H'], ['4', 'H'], ['5', 'H'], ['6', 'H'], ['7', 'H'],
    ['8', 'H'], ['9', 'H'], ['10', 'H'], ['J', 'H'], ['Q', 'H'], ['K', 'H'],
    ['A', 'S'], ['2', 'S'], ['3', 'S'], ['4', 'S'], ['5', 'S'], ['6', 'S'], ['7', 'S'],
    ['8', 'S'], ['9', 'S'], ['10', 'S'], ['J', 'S'], ['Q', 'S'], ['K', 'S']]
```

# Poker Hands: alternative representation

```
deck = [
    ['A', 'C'], ['2', 'C'], ['3', 'C'], ['4', 'C'], ['5', 'C'], ['6', 'C'], ['7', 'C'],
    ['8', 'C'], ['9', 'C'], ['10', 'C'], ['J', 'C'], ['Q', 'C'], ['K', 'C'],
    ['A', 'D'], ['2', 'D'], ['3', 'D'], ['4', 'D'], ['5', 'D'], ['6', 'D'], ['7', 'D'],
    ['8', 'D'], ['9', 'D'], ['10', 'D'], ['J', 'D'], ['Q', 'D'], ['K', 'D'],
    ['A', 'H'], ['2', 'H'], ['3', 'H'], ['4', 'H'], ['5', 'H'], ['6', 'H'], ['7', 'H'],
    ['8', 'H'], ['9', 'H'], ['10', 'H'], ['J', 'H'], ['Q', 'H'], ['K', 'H'],
    ['A', 'S'], ['2', 'S'], ['3', 'S'], ['4', 'S'], ['5', 'S'], ['6', 'S'], ['7', 'S'],
    ['8', 'S'], ['9', 'S'], ['10', 'S'], ['J', 'S'], ['Q', 'S'], ['K', 'S']]
```

```
FACE_VALUES = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10',
               'J', 'Q', 'K']
SUITS = ['C', 'D', 'H', 'S']

deck = []
for suit in SUITS :
    for face_value in FACE_VALUES :
        deck.append([face_value, suit])
```

# Multidimensional lists:

What is this? [ [ 'X', 'O', " " ], [ 'O', 'X', 'O' ], [ " ", " ", 'X' ] ]

# Multidimensional lists:

What is this? [ [ 'X', 'O', " " ], [ 'O', 'X', 'O' ], [ " ", " ", 'X' ] ]

When the sizes of **all** the lists are the same, this is called a

- multi-dimensional list
- in this case it is a 2D-list or array

Note: both outer and inner lists have 3-item in them.

This 2D lists represents the state of a Tic-Tac-Toe game

```
[['X', 'O', ' '],  
 ['O', 'X', 'O'],  
 [' ', ' ', 'X']]
```

X		O	
-----+-----+-----			
0		X	
-----+-----+-----			
			X

# Multidimensional lists:

```
define g = [ [ 'X', 'O', '' ], [ 'O', 'X', 'O' ], [ '', '', 'X' ] ]
```

g[0][0]	g[0][1]	g[0][2]
g[1][0]	g[1][1]	g[1][2]
g[2][0]	g[2][1]	g[2][2]

X		0	
-----+-----			
0		X	
-----+-----			
			X

```
if g[0][0]==g[0][1] and g[0][1]==g[0][2]:  
    print(g[0][0], 'has won!')  
else:  
    print('No one has won in the top row.')
```

```
if g[0][0]==g[1][1] and g[1][1]==g[2][2]:  
    print(g[0][0], 'has won!')  
else:  
    print('No one has won on the main diagonal.')
```

# String processing: Pallindromes

## Problem

Write a program that inputs a string and determines if it is a palindrome. A palindrome is a string that reads the same forwards or backwards, e.g. "bob" and "madam".

An excellent program would be able to deal with entire phrases by ignoring capitalization, spaces and punctuation in the input, e.g. "*A man, a plan, a canal, Panama!*" is a palindromic phrase and should be identified as such.

*For the moment, ignore phrases and focus on words only:*

*How can we determine that a word is a pallindrome?*

# String processing: Pallindromes (Solution 1)

A pallindrome reads the same backwards and forwards:  
1<sup>st</sup> letter forwards then should be the 1<sup>st</sup> letter backwards, i.e the last letter etc

```
if first_letter == last_letter and second_letter == second_last_letter  
    Then it is a palindrome.  
else  
    It is NOT a palindrome.
```

Python'ize this pseudocode:

```
if s[0]==s[len(s)-1] and s[1]==s[len(s)-2] and s[2]==s[len(s)-3] :  
    Then it is a palindrome.  
else  
    It is NOT a palindrome.
```

any problems here?

# String processing: Pallindromes (Solution 1)

We don't know ahead of time how many test expressions we need (because we don't know ahead of time how long the string is).

So use a **loop** instead of **if**

```
97 palindrome = True
98 for offset in range(0, len(s)//2):
99     if s[offset] != s[len(s)-1-offset]:
100         palindrome = False
101         break
102
103 if palindrome:
104     print("It is a palindrome!")
105 else:
106     print("It is NOT a palindrome.")
```

# String processing: Pallindromes (Solution 2)

Python must have a built-in library that we can leverage to solve this problem!

Another approach: build the reverse of the string, and compare:

Lists have a built-in reverse method, but strings don't.  
so, lets convert the string to a list first.

```
>>> s = "madam"
>>> slist = list(s)
>>> slist
['m', 'a', 'd', 'a', 'm']
>>>
```

Now we can reverse

```
>>> slist.reverse()
>>> slist
['m', 'a', 'd', 'a', 'm']
>>>
```

Note we can't do: `slist == slist.reverse()`  
since **.reverse()** changes `slist` in place,  
that is instead of creating a new string in reverse  
it modified `slist`!

# String processing: Pallindromes (Solution 2)

we can't do `slist == slist.revers()`, instead convert back to string and compare!  
For that purpose use the **.join()** method provided by string module

```
>>> iplist = ['199', '147', '23', '5']
>>> ip = '.'.join(iplist)
>>> ip
'199.147.23.5'
>>>
```

join items iplist with '.' in b/w

In order to join char's nothing in between use "":

```
>>> s_reversed = ''.join(slist)
>>> s_reversed
'madam'
>>>
```

# String processing: Pallindromes (Solution 2)

we can't do `slist == slist.revers()`, instead convert back to string and compare!  
For that purpose use the `.join()` method provided by string module

```
109 s = "madam"
110 slist = list(s)
111 slist.reverse()
112 s_reversed = ''.join(slist)
113 if s == s_reversed:
114     print("It is a palindrome!")
115 else:
116     print("It is NOT a palindrome.")
117
```

# String processing: Pallindromes (Testing)

So far we have just used "madam" and been happy when our code correctly identified it as a palindrome, but that's not sufficient testing. What would be good additional tests?

# String processing: Pallindromes (Testing)

So far we have just used "madam" and been pleased when our code correctly identified it as a palindrome, but that's not sufficient testing. What would be good additional tests?

- pallindromes with even number of letters
- non-palindromes with even & odd # of letters
- edge cases: zero and one-letter strings
- two letter strings: both palindrome & non-pallindrome
- non-alphabetic characters

A good set for testing ->

madam

maam

motor

moor

a

oo

at

A man, a plan, a canal, Panama!

# String processing: Pallindromes (Testing)

Poor man's testing:

```
130## palindrome: poor man's testing
131TESTS = ['madam', 'maam', 'motor', 'moor', 'a', 'oo', 'at',
132          'A man, a plan, a canal, Panama!']
133print 'Testing Solution 1:'
134for s in TESTS:
135    print s,
136    # Preprocess s to lower case and remove non-alphas.
137    ...
138
139    # Test to see if s is a palindrome
140    ...
```

Although this code does the job, it is an awkward way of doing testing our code because we had to modify our code substantially to test it.

Python allows us to make our palindrome testing code a stand alone function, and then to embed it into a module that does automatic testing when run on its own, but from which we can import the palindrome function if we need to use it.

Creating such modules and functions will be covered in the next class.

# Palindromes: Phrases

```
## palindrome: preprocessing for phrases
s = "A man, a plan, a canal, Panama."
print(s)
s_new = ''
for c in s:
    if c.isalpha():
        if c.isupper():
            s_new = s_new + c.lower()
        else:
            s_new = s_new + c
print("becomes")
print(s_new)
```

- convert all characters to lowercase
- get rid of punctuation & space etc

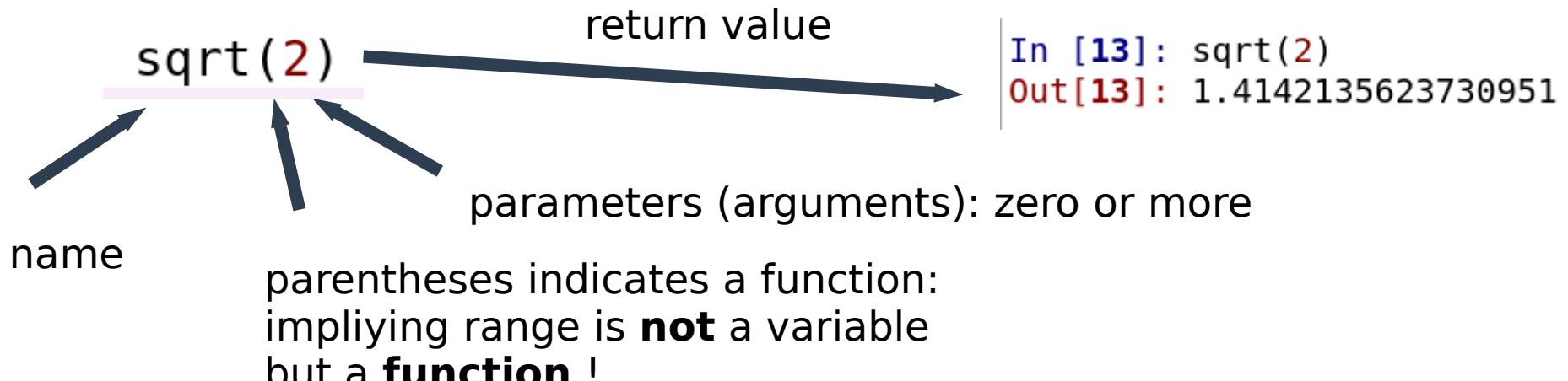
```
A man, a plan, a canal, Panama.
becomes
amanaplanacanalpanama
```

# Modularization

- So far we've seen key essential concepts of programming: input, processing, output, sequence, selection & repetition
- it is impossible to write any program without these
- **modularization** is another key concept
- although its optional, it is indeed essential
- examples:
  - **functions, classes, and modules**
- modularization is important because:
  - **reduces complexity**
  - enables **creation of reusable fragments of code**
- Reducing complexity is important:
  - divide a big programming task into smaller, more manageable pieces
  - carefully created modules can be reusable
  - human brain can't focus on more than a handful things at a time

# Functions

- We've already used many built-in functions in python:
- range(), math.sqrt(), random.randint()



- return value can be **None**
- there can be as few as zero number of arguments, **still () are required!**

# Writing functions: `is_even()`

Write a function by starting with how we want to use it.

Imagine that we're working on a game, and wish we could write:

```
num = random.randint(0,100)
if is_even(num):
    print("Good news your magic number is even!\n
        10 bonus points for you.")
else:
    print("Bad news, your magic number is odd.")
```

specifying the use of our function before writing it is that we now know,

- name of the function should be `is_even()`
- it takes only one parameter, `num`
- returns a Boolean value, `True` or `False`

From previous experience, we know how  
to implement a piece of code determining if  
a number is even or odd



```
if n%2 == 0:
    # It's even
else:
    # It's odd
```

# Writing functions: `is_even()`

- Keyword **def** marks the beginning of a function,
- followed by function name, parenthesis, parameters, and “**:**”
- code to be executed is indented
- **return** statement terminates the execution of function
- more than one **return** is possible
- the names of the parameter in the program and in the function don't need to be the same:  
**num** in the program is referred to as **n** in the function definition!
- What is the point? why not replace “if `is_even(num):`” with “if `num% == 0:`”?

```
import random

def is_even(n):
    if n%2 == 0:
        return True
    else:
        return False

num = random.randint(0,100)
if is_even(num):
    print("Good news your magic number is even!\n
          10 bonus points for you.")
else:
    print("Bad news, your magic number is odd.")
```

# Writing functions: `is_even()`

- What is the point? why not replace “if `is_even(num):`” with “`if num% == 0:`”?
- function is reusable by other programs!
- if name of the parameter in func definition and in the program were required to be the same, it would be very hard to reuse that function!
- **n** is an alias to value of **num**



- as soon as function is terminated, any memory it used is freed
- hence variables, constants, etc in a function definition are local to that function!

```
import random

def is_even(n):
    if n%2 == 0:
        return True
    else:
        return False

num = random.randint(0,100)
if is_even(num):
    print("Good news your mag"
          "10 bonus points for you.")
else:
    print("Bad news, your mag")
```

# Writing functions: Syntax Summary

```
def function-name( parameter-list ):  
    ...  
    ... function-body  
    ...  
    return expression  
    ...  
    return expression
```

- it is preferable to have a single **return** expression or value,
- and it is preferable to have it at the last line of the function
- if no **return** is included, function will return **None**
- it is preferable to have a blank **return** than having none.  
(showing our intention of returning None as opposed to forgetting to write a return expression)

# Example: is\_odd()

```
def is_odd(n):
    if n%2 != 0:
        return True
    else:
        return False
```

is there a better way?

# Example: is\_odd()

```
def is_odd(n):  
    return not is_even(n)
```

```
def is_odd(n):  
    if n%2 != 0:  
        return True  
    else:  
        return False
```

- less lines of code
- reuse
- if someday a better way of determining is\_even() is found
  - only one function needs to rewritten
  - improvements in is\_even() automatically passed on to is\_odd(): this case is trivial, but in many computationally complex programs, this happens all the time!

# Programming Exercise: Dice\_roll()

Write a program rolling a pair of dice and returning the total.  
Modularize dice roll section of the code via use of a function like:

```
total = dice_roll() + dice_roll()
```

# Programming Exercise: Dice\_roll()

```
import random

def dice_roll():
    return random.randint(1,6)

total = dice_roll() + dice_roll()
print("On your first roll you got:", total)
```

# Programming Exercise: Dice\_roll()

```
import random

def dice_roll():
    return random.randint(1,6)

total = dice_roll() + dice_roll()
print("On your first roll you got:", total)
```

What if there is more than 6 sides? Can we generalize this?



# Programming Exercise: Dice\_roll()

```
import random

def dice_roll(sides):
    return random.randint(1,sides)

total = dice_roll(6) + dice_roll(6)
print("On your first roll you got:", total)
```

# Default arguments

```
import random

def dice_roll(sides):
    return random.randint(1,sides)

total = dice_roll(6) + dice_roll(6)
print("On your first roll you got:", total)
```

- we improved dice rolling function via generalizing it to more than 6 sides
- however for the most common case, 6 sides, the usage got a bit complicated because now we have to specify number of sides!
- To avoid this complication, we can specify default values of arguments:  
**default value**: a value to be used when no value is given/specifyed.

# Default arguments

```
import random

def dice_roll(sides = 6):
    return random.randint(1,sides)

print('Your results are:')
print('6-sided die:', dice_roll())
print('24-sided die:', dice_roll(24))
```

yields:

```
.....
Your results are:
6-sided die: 1
24-sided die: 15
```

default argument values are very common in Python

```
119
120 range()
121
122     Arguments
123         range(start, stop=None, step=1)
124
```

# Example: playing card functions

Let's use functions ( and later on classes) to save time when working with playing cards.

Create functions for common tasks: getting suit & face value from card number

```
SUITS = ('Clubs', 'Diamonds', 'Hearts', 'Spades')
FACE_VALUES = ('Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
                'Seven', 'Eight', 'Nine', 'Ten', 'Jack',
                'Queen', 'King')

def suit(cardnum):
    return SUITS[cardnum // 13]

def face_value(cardnum):
    return FACE_VALUES[cardnum % 13]

card = 15
print("Card", card, "is the", face_value(card), "of", suit(card))
```

# Example: playing card functions

We will also need to display card label very often ("Three of diamonds" etc)  
So, let's create a function for that too

```
SUITS = ('Clubs', 'Diamonds', 'Hearts', 'Spades')
FACE_VALUES = ('Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
               'Seven', 'Eight', 'Nine', 'Ten', 'Jack',
               'Queen', 'King')

def suit(cardnum):
    return SUITS[cardnum // 13]

def face_value(cardnum):
    return FACE_VALUES[cardnum % 13]

def label(cardnum):
    return face_value(cardnum) + " of " + suit(cardnum)

card = 15
print("Card", card, "is the", face_value(card), "of", suit(card))
print("Card", card, "is the", label(card))
```

output is:

```
Card 15 is the Three of Diamonds
Card 15 is the Three of Diamonds
```

# Reusing functions

Now that we've written some reusable functions dealing with playing cards, how do we go about reusing them?

- imagine these functions are needed in a mobile app, and in a webapp
- Option #1: copy & paste these functions to mobile-app and webapp programs
- What can go wrong?

# Reusing functions

Now that we've written some reusable functions dealing with playing cards, how do we go about reusing them?

- imagine these functions are needed in a mobile app, and in a webapp
- Option #1: copy & paste these functions to mobile-app and webapp programs
  - improve these functions, **re-copy** & paste them to mobileapp, webapp,
    - a bug found and fixed. Where was that up-to-date list of all programs using these functions?
    - it would be nice if there was a way to automate this process, so that programs always benefit from improvements to these functions.

**What would that be?**

# Reusing functions

Now that we've written some reusable functions dealing with playing cards, how do we go about reusing them?

- Programmers found that the best way to reuse code is to include the contents of one file in another
- Python does this by using **modules** and the **import** command
- We've already used **import** to use functions from **math** and **random** modules
- Now let's see how we can do this for our own functions

# Modules

- A python program is a module
- and other programs can use the functions, variables, etc in it
- but we need to follow some standard practices for the code to be reused easily
- Lets put playing card functions in **playing\_cards.py**
- our new program **blackjack.py** have a hand of card numbers
  - we would like to display the cards in the hand:

```
# blackjack.py
...
print 'You are holding,'
for card in hand:
    print 'The', label(card)
...
```

expected  
output



You are holding,  
The Three of Diamonds  
The Four of Spades  
The Nine of Clubs

# Modules: First Try

```
# blackjack.py
import playing_cards

hand = [15, 42, 8]
print('You are holding, ')
for card in hand:
    print('The', playing_cards.label(card))
```

Note, when importing we omit \*.py extension

just as we didn't do "math.py.sqrt" but **math.sqrt**

What do you think we should get? [demo]

# Modules: A problem

```
>>>  
Card 15 is the Three of Diamonds  
Card 15 is the Three of Diamonds  
You are holding,  
The Three of Diamonds  
The Four of Spades  
The Nine of Clubs  
>>>
```

- we are getting extraneous output from **playing\_cards.py**
- modules are run upon import!
- Code is interpreted and definitions in the module become available to current program
- Should we eliminate print statements? What if they're necessary?

# Modules: A solution

- Every module has a `_name_` attribute
- When run on its own `_name_` is assigned the name `_main_`
- When imported, `_name_` is assigned the name of the module: `playing_cards`
- Lets run the following, and see what we get [demo]

```
14
15 def suit(cardnum):
16     return SUITS[cardnum // 13]
17
18 def face_value(cardnum):
19     return FACE_VALUES[cardnum % 13]
20
21 def label(cardnum):
22     return face_value(cardnum) + " of " + suit(cardnum)
23
24 print('My name is', __name__) # Line 24!
25 card = 15
26 print("Card", card, "is the", face_value(card), "of", suit(card))
27 print("Card", card, "is the", label(card))
```

# Modules: A solution

When we run **playing\_cards.py**, we get

```
My name is __main__  
Card 15 is the Three of Diamonds  
Card 15 is the Three of Diamonds
```

When we run **blackjack.py**, we get

```
My name is playing_cards  
Card 15 is the Three of Diamonds  
Card 15 is the Three of Diamonds  
You are holding,  
The Three of Diamonds  
The Four of Spades  
The Nine of Clubs
```

# Modules: A solution

When we run **playing\_cards.py**, we get

```
My name is __main__  
Card 15 is the Three of Diamonds  
Card 15 is the Three of Diamonds
```

When we run **blackjack.py**, we get

```
My name is playing_cards  
Card 15 is the Three of Diamonds  
Card 15 is the Three of Diamonds  
You are holding,  
The Three of Diamonds  
The Four of Spades  
The Nine of Clubs
```

Solution:

put an **if** statement:

only if `__name__ == "__main__"`

then print stuff

Hence, when imported by a program  
we won't see these extra lines!

# Modules: A solution

```
SUITS = ('Clubs', 'Diamonds', 'Hearts', 'Spades')
FACE_VALUES = ('Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
                'Seven', 'Eight', 'Nine', 'Ten', 'Jack',
                'Queen', 'King')

def suit(cardnum):
    return SUITS[cardnum // 13]

def face_value(cardnum):
    return FACE_VALUES[cardnum % 13]

def label(cardnum):
    return face_value(cardnum) + " of " + suit(cardnum)

if __name__ == '__main__':
    card = 15
    print("Card", card, "is the", face_value(card), "of", suit(card))
    print("Card", card, "is the", label(card))
```

# Documenting modules and functions

So far we did:

- block of header comments identifying module, programmer, date etc
- inline comments to explain tricky points

However, modules and functions require further documentation for which primary mechanism is docstrings, i.e. the triple quote text

```
# module_docn.py
'''This is the module documentation pointing out that
this is an artificial test module.'''

def test_fn_1():
    '''This is the first test function.
    It doesn't do anything.'''
    return

def test_fn_2():
    '''This is the second test function which also does nothing.'''
    return
```

# Documenting modules and functions

Python has builtin commands that extract this documentation from module

```
In [4]: import module_docn  
  
In [5]: dir(module_docn)  
Out[5]:  
['__builtins__',  
 '__cached__',  
 '__doc__',  
 '__file__',  
 '__loader__',  
 '__name__',  
 '__package__',  
 '__spec__',  
 'test_fn_1',  
 'test_fn_2']
```

```
In [6]: print(module_docn.__doc__)  
This is the module documentation pointing out that  
this is an artificial test module.  
  
In [7]: print(module_docn.test_fn_1.__doc__)  
This is the first test function.  
It doesn't do anything.  
  
In [8]: print(module_docn.test_fn_2.__doc__)  
This is the second test function which also does nothing.
```

docstrings: strings that get assigned to the `__doc__` attributes

# Documenting modules and functions

An easy way to access docstrings is to use **help()**

```
In [10]: help(module_docn)
Help on module module_docn:

NAME
    module_docn

DESCRIPTION
    This is the module documentation pointing out that
    this is an artificial test module.

FUNCTIONS
    test_fn_1()
        This is the first test function.
        It doesn't do anything.

    test_fn_2()
        This is the second test function which also does nothing.

FILE
    /home/sbulut/github/cpsc128/code/python3/module_docn.py
```

# Summary: Module layout

```
# filename.py
# ...
''' Module docstring '''

# import statements
import ...
import ...
...

# Function definitions
def name( args ):
    ''' Function docstring '''
    ...

def name( args ):
    ''' Function docstring '''
    ...

...
if __name__ == '__main__':
    # Print statements, simple examples etc
    ...
```

# Demo: playing\_cards\_2

- See playing\_cards\_2.py
- run it, and observe the output
- also import it
- also look at help() page

```
In [16]: runfile('/home/sbulut/github/cpsc128/code/python3/  
playing_cards_2.py', wdir='/home/sbulut/github/cpsc128/code/python3')  
New deck: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,  
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,  
37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51]  
Shuffled deck: [51, 7, 43, 34, 2, 39, 14, 8, 40, 42, 45, 15, 35, 48, 13,  
17, 19, 36, 26, 31, 38, 16, 21, 22, 30, 3, 41, 27, 0, 24, 11, 23, 32, 12,  
46, 6, 20, 37, 47, 5, 50, 9, 44, 18, 49, 10, 29, 4, 33, 25, 1, 28]  
Dealing a card...  
    Card number: 28  
    Face value: Three  
    Suit: Hearts  
    Description: Three of Hearts  
Deck after dealing: [51, 7, 43, 34, 2, 39, 14, 8, 40, 42, 45, 15, 35, 48,  
13, 17, 19, 36, 26, 31, 38, 16, 21, 22, 30, 3, 41, 27, 0, 24, 11, 23, 32,  
12, 46, 6, 20, 37, 47, 5, 50, 9, 44, 18, 49, 10, 29, 4, 33, 25, 1]
```

# import tricks

```
card=10

import playing_cards
print(playing_cards.label(card))

import playing_cards as pc
print(pc.label(card))

from playing_cards import label
print(label(card))

from playing_cards import *
print(label(card))

from playing_cards import label as card_name
print(card_name(card))
```

# Scope of variables ( a technical issue)

Let's analyze this snippet

```
...
SUITS = ('Clubs', 'Diamonds', 'Hearts', 'Spades')
...

def suit(cardnum):
    '''Returns the suit name, e.g. 'Clubs', of the card it is passed.'''
    return SUITS[cardnum // 13]

def suit(cardnum):
    SUITS = ('Spades', 'Hearts', 'Diamonds', 'Clubs' )
    '''Returns the suit name, e.g. 'Clubs', of the card it is passed.'''
    return SUITS[cardnum // 13]
```

checkout demo:  
scope\_eg.py

Python functions follow a clear process in trying to resolve names.

- Python first looks in the **Local** context, i.e. the current function.
- Then it looks in any **Enclosing** functions, i.e. functions that contain the current one.
- Then it looks in the **Global** context, i.e. the module.
- Finally it checks the **Built-in** names for a match.

**Be careful, when local definitions are overriding globals,  
or globals overriding built-in ones**

# Perils of mutability (a technical issue)

```
# increment.py
def increment(n):
    n = n + 1

num = 48
increment( num )
print(num)
```

```
def increment(seq):
    seq.append(42)

lst = [48]
increment( lst )
print(lst)
```

What is the output?

# Perils of mutability (a technical issue)

Unlike numbers and strings, lists are mutable so they can be affected when passed to functions, e.g.

```
l = [ 'Tim' ]
n = l
l.append( 'Joyce' )
print(l)
print(n)
```

```
a = 'Tim'
b = 'Tom'
lst = [b, a]
a = 'Matt'
lst[1] = lst[0]
print(lst)
```

Don't get tripped by mutability!

# Copying a list: deep vs shallow

For a shallow copy: do

```
In [63]: l1 = [1, 2]
In [64]: l2 = list(l1) ; l3=l2[:]
In [65]: l2[0] = 3 ; l3[1] = 4
In [66]: print(l1); print(l2) ; print(l3)
[1, 2]
[3, 2]
[1, 4]
```

Shallow copy sort of works  
for list of lists

```
In [97]: g =[ [1,2], 3]
In [98]: c = list(g)
In [99]: c.append('hello')
In [100]: print(g); print(c)
[[1, 2], 3]
[[1, 2], 3, 'hello']

In [101]: c[0][0] = 10
In [102]: print(g); print(c)
[[10, 2], 3]
[[10, 2], 3, 'hello']
```

# Copying a list: deep vs shallow

Solution: use **deepcopy** from **copy** module

```
In [112]: import copy  
  
In [113]: g = [ [1,2], 3]  
  
In [114]: dc = copy.deepcopy(g)  
  
In [115]: dc[0][0] = 'x'  
  
In [116]: print(g)  
[[1, 2], 3]  
  
In [117]: print(dc)  
[['x', 2], 3]
```

# More aggregate data types: Dictionaries

- another useful data type in Python
- unordered set of **key:value** pairs

```
In [6]: x = {} # here is an empty dict  
  
In [7]: type(x)  
Out[7]: dict  
  
In [8]: d = { 'Sinan' : 705, 'Brian' : 867 }  
  
In [9]: d['Sinan']  
Out[9]: 705
```

- sequences: indexed by numbers
- dictionary: indexed by keys
- keys must be (i) of immutable type and (ii) unique!!

# Aside: known aliases to dictionaries

- Dictionary: Python, Smalltalk, Objective-C, .NET.
- **Hash or hash table**: C, C++, Perl, Ruby, Visual Basic, Common Lisp.
- **Collection**: Visual Basic
- **Map**: C++, Java.
- **Associative array**: Javascript, AWK

# Why use dictionaries?

- dictionary lends itself naturally to a mapping situation

example: a pair of dice roll odds

```
16 d = {2:      29,
17     3:      66,
18     4:      71,
19     5:      100,
20     6:      137,
21     7:      173,
22     8:      132,
23     9:      122,
24    10:      78,
25    11:      59,
26    12:      33
27 }
28
29 rollsum = dice_roll() + dice_roll()
30
31 d[rollsum] = d[rollsum] + 1
```

	<b>Dice outcome</b>	<b>Counter</b>
	2	→ counters[0]
	3	→ counters[1]
	4	→ counters[2]
	...	...
	11	→ counters[9]
	12	→ counters[10]

# Why use dictionaries?

- performance wise dictionaries are absolutely superior to lists in look up situations where there is no simple way to map key to values
  - dictionary
  - flight number + day -> arrival time
  - name -> phone number
- in a look up scenario, computational time
  - increases linearly as a function of problem size (N) in **lists**
  - nearly independent of problem size in **dicts** !!!

# Working with dictionary type

```
len(d)          # returns number of items in d
d.keys()        # returns a view of keys in d
d.values()      # returns a view of values in d
k in d          # returns True if k in d
d[k]            # returns value associated with key k
d[k] = v         # associates value v with key k
del d[k]         # removes entry with key k from d
d.pop(k)         # remove entry with key k but get value
for k in d:     # iterate over keys in d
```

# Example: Word frequencies

Given an input string, display

- i) each word and
- ii) how many times it occurs in the string

```
1# WordFrequencies.py
2# Get the string:
3test_string = '''The programmer, who needs clarity, who must talk all day to
4    a machine that demands declarations, hunkers down into a low-grade annoyance.
5    It is here that the stereotype of the programmer,
6    sitting in a dim room, growling from behind Coke cans, has its origins.
7    The disorder of the desk, the floor; the yellow Post-It notes everywhere;
8    the whiteboards covered with scrawl:
9    all this is the outward manifestation of the messiness of human thought.
L0    The messiness cannot go into the program;
L1    it piles up around the programmer.
L2    ~ Ellen Ullman
L3'''
```

# Example: Word frequencies

Given an input string, display each word and how many times it occurs in the string

Pseudocode

```
get the string
break the string into a list of words
initialize a dictionary of counters
loop through the list of words
    if the word is in the dictionary
        increment its counter by 1
    otherwise
        set its counter to 1
loop through the entries in the dictionary
    for each entry display the key (the word) and the value
```

# Example: Word frequencies

```
1 # WordFrequencies.py
2 # Get the string:
3 test_string = '''The programmer, who needs clarity, who must talk all day to
4     a machine that demands declarations, hunkers down into a low-grade annoyance.
5     It is here that the stereotype of the programmer,
6     sitting in a dim room, growling from behind Coke cans, has its origins.
7     The disorder of the desk, the floor; the yellow Post-It notes everywhere;
8     the whiteboards covered with scrawl:
9     all this is the outward manifestation of the messiness of human thought.
10    The messiness cannot go into the program;
11    it piles up around the programmer.
12    ~ Ellen Ullman
13 ...
14
15 # Break the string into a list of words:
16 words = test_string.split()
17 # Initialize a dictionary of counters:
18 word_counts = {}
19
20 # Loop through the list of words:
21 for word in words:
22     # If the word is in the dictionary:
23     if word in word_counts:
24         # Increment its counter by 1:
25         word_counts[word] = word_counts[word] + 1
26     # Otherwise:
27     else:
28         # Set its counter to 1:
29         word_counts[word] = 1
30
31 # Loop through the entries in the dictionary:
32 for word in word_counts:
33     # Display the key value pair:
34     print(word, ':', word_counts[word])
```

[demo]

# Example: Word frequencies improvements

problems:

The : 3

floor; : 1

annoyance. : 1

the : 10

~ : 1

What string methods can we use to fix these?

# Example: Word frequencies improvements

problems:

The : 3

floor; : 1

annoyance. : 1

the : 10

~ : 1

```
37 for word in words:  
38     w = word.lower()  
39     w = w.strip('.;,:\'"?!)') # Notice the \ to escape  
40     if w.isalpha():  
41         if w in word_counts:  
42             word_counts[w] = word_counts[w] + 1  
43         else:  
44             word_counts[w] = 1  
45
```

# Example: Scrabble scoring

Write a function taking a word as input, and returning the total Scrabble value.

Use a dictionary to map letters to their scrabble value

```
A=1 B=3 C=3 D=2 E=1 F=4 G=2 H=4 I=1 J=8 K=5 L=1 M=3  
N=1 O=1 P=3 Q=10 R=1 S=1 T=1 U=1 V=4 W=4 X=8 Y=4 Z=1
```

Pseudocode ?

# Example: Scrabble scoring

Write a function taking a word as input, and returning the total Scrabble value.

Use a dictionary to map letters to their scrabble value

A=1	B=3	C=3	D=2	E=1	F=4	G=2	H=4	I=1	J=8	K=5	L=1	M=3
N=1	O=1	P=3	Q=10	R=1	S=1	T=1	U=1	V=4	W=4	X=8	Y=4	Z=1

Pseudocode

```
def ScrabbleValue( s ):  
    Initialize the total value to 0  
    Loop through the word a letter at a time  
        Look up the letter's value  
        Increment the total value by this letter's value  
    Return the total value
```

# Example: Scrabble scoring

```
1 # ScrabbleScoring.py
2 LETTER_VALUES = {'A':1, 'B':3, 'C':3, 'D':2, 'E':1, 'F':4, 'G':2,
3                 'H':4, 'I':1, 'J':8, 'K':5, 'L':1, 'M':3, 'N':1,
4                 'O':1, 'P':3, 'Q':10, 'R':1, 'S':1, 'T':1, 'U':1,
5                 'V':4, 'W':4, 'X':8, 'Y':4, 'Z':1}
6
7 def scrabble_value(s):
8     total_value = 0
9     for letter in s:
10         total_value = total_value + LETTER_VALUES[letter]
11     return total_value
12
13 if __name__ == '__main__':
14     print('The value of the word HERE is', scrabble_value('HERE'))
```

# Example: Book database

- This is an example where values are non-numerical
- A small database of book information stored in a dictionary
- Key: book title, value: authors
- There can be more than one author, so we're using lists
- Write a function listing titles of the books by a given author!

```
books = { "The C Programming Language": ['Brian W. Kernighan',
                                             'Dennis M. Ritchie'],
          "Harry Potter and the Philosopher's Stone" : ['J. K. Rowling'],
          "The AWK programming language" : ['Alfred V. Aho',
                                             'Brian W. Kernighan',
                                             'Peter J. Weinberger'],
          "The practice of programming": ['Brian W. Kernighan', 'Rob Pike']
          "The cat in the hat": ['Dr. Seuss'],
          "The UNIX Programming Environment": ['Brian W. Kernighan',
                                              'Rob Pike'],
        }
```

# Example: Book database

- Write a function listing titles of the books by a given author!

Pseudocode

```
def search_by_author( database, author ):  
    Initialize book_list (the list of books by this author)  
    For each key-value pair in the database  
        If the value contains author  
            Add the key to the book_list  
    Return book_list
```

```
books = { "The C Programming Language": ['Brian W. Kernighan',  
                                         'Dennis M. Ritchie'],  
          "Harry Potter and the Philosopher's Stone" : ['J. K. Rowling'],  
          "The AWK programming language" : ['Alfred V. Aho',  
                                           'Brian W. Kernighan',  
                                           'Peter J. Weinberger'],  
          "The practice of programming": ['Brian W. Kernighan', 'Rob Pike'],  
          "The cat in the hat": ['Dr. Seuss'],  
          "The UNIX Programming Environment": ['Brian W. Kernighan',  
                                              'Rob Pike'],  
          }  
      }
```

# Example: Book database

- Write a function listing titles of the books by a given author!

```
1 # BooksDict.py
2 def search_by_author(database, author):
3     book_list = []
4     for key in database:
5         if author in database[key]:
6             book_list.append( key )
7     return book_list
8
9 if __name__ == '__main__':
10    books = { "Harry Potter and the Philosopher's Stone" : ['J. K. Rowling'],
11              "The cat in the hat": ['Dr. Seuss'],
12              "The C Programming Language": ['Brian W. Kernighan',
13                                              'Dennis M. Ritchie'],
14              "The UNIX Programming Environment": ['Brian W. Kernighan',
15                                              'Rob Pike'],
16              "The AWK programming language" : ['Alfred V. Aho',
17                                              'Brian W. Kernighan',
18                                              'Peter J. Weinberger'],
19              "The practice of programming": ['Brian W. Kernighan', 'Rob Pike']
20          }
21    print('J. K. Rowling wrote:', search_by_author(books, 'J. K. Rowling'))
22    print('Brian W. Kernighan wrote:',search_by_author(books, 'Brian W. Kernighan'))
23    print('Sinan Bulut wrote', search_by_author(books, 'Tim Topper'))
```

# Example: List of Dictionaries

- Just as other types can occur in a dictionary, dictionaries can occur in other types
- the fact that we can combine different types is a powerful programming concept

```
data = [ {'id':4721, 'sex':'F', 'age':31},  
        {'id':1828, 'sex':'M', 'age':56},  
        {'id':7816, 'sex':'M', 'age':72},  
        #. . . lots more records . . .  
        {'id':3286, 'sex':'M', 'age':29},  
        {'id':5063, 'sex':'F', 'age':22}  
    ]
```

Problem 1: How can display the entry of the oldest individual?

# Example: List of Dictionaries

## problem 1

Problem 1: How can display the entry of the oldest individual?

Pseudocode

We have to start somewhere so let's begin with the first entry and set it to be the oldest record (after all it's the oldest we have seen so far!)

Consider each item in the database from the second to the end

If the age of this entry is older than our current oldest

    Update our oldest record

Display the oldest record

```
data = [ {'id':4721, 'sex':'F', 'age':31},  
        {'id':1828, 'sex':'M', 'age':56},  
        {'id':7816, 'sex':'M', 'age':72},  
        #. . . lots more records . . .  
        {'id':3286, 'sex':'M', 'age':29},  
        {'id':5063, 'sex':'F', 'age':22}]
```

# Example: List of Dictionaries

## problem 1

Problem 1: How can display the entry of the oldest individual?

```
1# ListOfDicts.py
2data = [ {'id':4721, 'sex':'F', 'age':31},
3        {'id':1828, 'sex':'M', 'age':56},
4        {'id':7816, 'sex':'M', 'age':72},
5        {'id':3286, 'sex':'M', 'age':29},
6        {'id':5063, 'sex':'F', 'age':22}
7    ]
8
9oldest = data[0]
10for entry in data[1:]:
11    if entry['age'] > oldest['age']:
12        oldest = entry
13print('The oldest person is:', oldest)
```

# Example: List of Dictionaries

## problem 1

Problem 2: output number of male and female records

```
data = [ {'id':4721, 'sex':'F', 'age':31},  
        {'id':1828, 'sex':'M', 'age':56},  
        {'id':7816, 'sex':'M', 'age':72},  
        #. . . lots more records . . .  
        {'id':3286, 'sex':'M', 'age':29},  
        {'id':5063, 'sex':'F', 'age':22}  
    ]
```

# Example: List of Dictionaries

## problem 2

Problem 2: output number of male and female records

```
data = [ {'id':4721, 'sex':'F', 'age':31},  
        {'id':1828, 'sex':'M', 'age':56},  
        {'id':7816, 'sex':'M', 'age':72},  
        #. . . lots more records . . .  
        {'id':3286, 'sex':'M', 'age':29},  
        {'id':5063, 'sex':'F', 'age':22}  
    ]
```

# Example: List of Dictionaries

## problem 2

Problem 2: output number of male and female records

Pseudocode

```
Set counter of males to 0
Set counter of females to 0
Consider each item in the database
    If the value for the key 'sex' is 'M'
        Increment the counter of males
    Elif the value of the key 'sex' is 'F'
        Increment the counter of females
Display the male and female counters
```

# Example: List of Dictionaries

## problem 2

Problem 2: output number of male and female records

```
18 nmales = 0
19 nfemales = 0
20 for entry in data:
21     if entry['sex'] == 'M':
22         nmales = nmales + 1
23     elif entry['sex'] == 'F':
24         nfemales = nfemales + 1
25 print('There are', nmales, 'males and', nfemales, 'females.')
```

Aside: it is a common practice to name counter variables with 'n' prefix

like **nfemales**, **nmales**

as a short from of phrase “**number of males**” etc.

# Persistence: working with text files

- so far our programs have been independent from rest
- they started with a blank slate of memory, and worked with input provided during that specific run of the program
- all results were lost when program ended
- This is for with calculation type programs
- Other types of programs work with existing data or need to store their results
- Example: Office software: we should be able to resume editing
- Example: Game soft: game state should be saved so that user can resume their game

# Persistence: working with text files

- Many software need ability to move data out of RAM and put in a lasting storage area, i.e. disk
- RAM storage is volatile: it disappears when program ends or when computer is turned off
- Disk storage is persistent
- Many software require 'data persistence'
- In this course, we will focus on storing data as:
  - local text file
  - local binary file
- Other options are: storing data in a database, or remotely

# **Text vs binary files**

- Anything on a disk is stored as 0's and 1's:  
**how come there are two types: text vs binary?**
- The difference is the way 0's 1's interpreted!
- Text files are human readable

# Reading from text files

```
$ cat > text_file.txt  
The first line.  
Line 2.  
The third and the last line.
```

we can visualize this on the disk as:

```
The first line. \n Line 2. \n The third and last line. \n EOF
```

The main difference between what we write and how its stored is `\n`  
`\n` denotes the **newline** character

There are three common ways of reading in a text file:

1. one line at a time (preferred)
2. the whole file into a string
3. the whole file into a list of strings, one entry per line

Option 1 is preferred since it occupies less memory as one line at a time kept in mem

# Reading from text files:

## 1. One line at a time

```
1# file_read_1.py
2f = open('text_file.txt', 'r') # Open the file.
3for line in f:
4    print(line, end='')
5f.close()
```



# Iterate through the file a line at a time.  
# Process the current line.  
# Close the file.

'r' for 'read' mode

- file object 'f' is iterable: hence can used in **for** statement
- default iteration is to grab one line at a time
- Always close files via **f.close()** when you're finished and hence release memory.

```
8# alternative with 'with'
9with open('text_file.txt', 'r') as f:
10    for line in f:
11        print(line, end='')
```

# Reading from text files:

## 2. The whole file into string

```
1# file_read_2.py
2f = open('text_file.txt', 'r')
3s = f.read()
4print('s is', len(s), 'characters long.')
5print(s)
6f.close()
```

```
In [41]: runfile('/home/sbulut
code/python3')
s is 53 characters long.
The first line.
Line 2.
The third and the last line.
```

```
In [42]:
```

- **read** method reads entire file as string including \n characters
- '\n' characters are interpreted, hence file looks same as reading one line at a time
- f.read(n) reads only n characters

# Reading from text files:

## 3. The whole file into list of strings

```
1# file_read_3.py
2f = open('text_file.txt', 'r')
3lines = f.readlines()
4print(lines)
5f.close()
```

Another option is:

```
lines = list(f)
```

```
In [46]: runfile('/home/sbulut/github/cpsc128/code/python3/file_read_
code/python3')
['The first line.\n', 'Line 2.\n', 'The third and the last line.\n']
```

Note the newline character. If you don't want it just strip it!

# Example: search log files

Suppose as part of a security audit we want to display all the lines from our web server log file containing the IP address 199.247.232.110. The file name is **access.log.1** and the first ten lines of the file look like this,

Typically such files are quite large 100's of Megabytes.  
So, read one line at a time!

## Pseudocode

```
Get the name of the log file
Get the name of the IP address to scan for
Open the file for reading
For each line in the file
    If the line contains the IP address
        Display the line
```

# Example: search log files

```
1# ip_extractor.py
2fname = input('What file do you want to scan? ')
3ip = input('What IP address do you want to scan for? ')
4print()
5f = open(fname, 'r')
6for line in f:
7    if line.find(ip) != -1:
8        print(line)
9
```

What file do you want to scan? access.log.1

What IP address do you want to scan for? 199.247.232.110

199.247.232.110 - - [12/Apr/2009:08:52:10 -0700] "GET /Math  
HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows; U; Windows NT 5.  
AppleWebKit/525.19 (KHTML, like Gecko) Chrome/1.0.154.53 Sa

199.247.232.110 - - [12/Apr/2009:08:53:00 -0700] "GET /Math  
HTTP/1.1" 200 2326 "http://ttopper.yukoncollege.yk.ca/Math]  
"Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/  
Gecko) Chrome/1.0.154.53 Safari/525.19"

# Files are sequential

In Python, a file's content are read in sequentially similar to the physical tapes.

```
>>> f = open('text_file.txt','r')
>>> s = f.read()
>>> s
'The first line.\nLine 2.\nThe third and last line.\n'
>>> p = f.read()
>>> p
''
```

# Files are sequential: other methods

```
In [69]: f = open('text_file.txt', 'r')
```

```
In [70]: f.tell()
```

```
Out[70]: 0
```

```
In [71]: f.read(10)
```

```
Out[71]: 'The first '
```

```
In [72]: f.tell()
```

```
Out[72]: 10
```

```
In [73]: f.read(10)
```

```
Out[73]: 'line.\nLine'
```

```
In [78]: f.seek(10)
```

```
Out[78]: 10
```

```
In [79]: f.read()
```

```
Out[79]: 'line.\nLine 2.\nThe third and the last line.\n'
```

```
In [80]: f.closed
```

```
Out[80]: False
```

# Writing to text files

Suppose we need to store a list of coordinates to a text file

```
coords = [[12, 31], [75, 19], [28, 51]]
```

```
1 # write_coords.py
2 coords = [[12, 31], [75, 19], [28, 51]]
3 fname = input('Name of file to create? ')
4 f = open(fname, 'w')
5 for coord in coords:
6     f.write(str(coord[0]) + ' ' + str(coord[1]) + '\n')
7 f.close()
```

- Note ‘w’ for ‘write’: we open file in write mode
- Opening a file in write mode deletes a file if it already exists!
- We direct output to file using **write** method
- **write** accepts **str** type as argument
- Hence convert the list to string, and also add \n
- remember to close the file

# Appending to files

- What if we want to append to a file?
- We can't use **write** since it destroys a file
- So use 'a' (append) mode instead!

```
In [106]: fname='out.txt'

In [107]: f = open(fname, 'w')

In [108]: f.write("hello\n")
Out[108]: 6

In [109]: f = open(fname, 'a')

In [110]: f.write("world\n")
Out[110]: 6

In [111]: with open(fname, 'r') as f:
...:     print(f.read())
...:
hello
world
```

# Reading in numerical data

Open the file

Initialize coords to an empty list

Read it a line at a time

    Split the line into parts at blanks

    Convert each part into an integer value

    Append the integer values to the list coords

# Reading in numerical data

```
1# read_coords.py
2coords = []
3fname = input('Name of file to read from? ')
4f = open(fname, 'r')
5for line in f:
6    x_string, y_string = line.split()
7    coords.append([int(x_string),int(y_string)])
8f.close()
9print('coords =', coords)
```

Note the multiple assignment!

Aside:

```
1# read_coords_v2.py
2coords = []
3fname = input('Name of file to read from? ')
4f = open(fname, 'r')
5for line in f:
6    coords.append( list( map(int,line.split()) ) )
7f.close()
8print('coords =', coords)
9
```

# Designing File Formats

- In the previous example, we stored coordinates to: one pair per line separated by single space
- instead, we could have put all in one line and separate items by commas
- one line vs many lines, spaces vs commas etc are in this case cosmetic
- Sometimes different approaches can be quite different in nature

# Conway's game of life!

A simple, fun problem that will show us pros/cons of different data storage approaches

The universe in Conway's game of life is a grid of cells each of which can be in one of two states: alive or dead. A natural way to represent it in Python would be as a list of lists of cells (like a very large tic-tac-toe board):

```
universe = [ [0, 0, 0, 0, 0, 0, 0, 0],  
             [0, 0, 0, 0, 0, 0, 0, 0],  
             [0, 0, 0, 1, 0, 0, 0, 0],  
             [0, 0, 0, 0, 1, 0, 0, 0],  
             [0, 0, 1, 1, 1, 0, 0, 0],  
             [0, 0, 0, 0, 0, 0, 0, 0],  
             [0, 0, 0, 0, 0, 0, 0, 0],  
             [0, 0, 0, 0, 0, 0, 0, 0]  
         ]
```

How can we put this in a text file?

# Storing universe in Conway's game of life

## Option 1:

- space separated 0's 1's
- storing each character costs 1 byte (8bits)
- 64 0/1 characters = 64 bytes
- 64 spaces to separate them = 64 bytes
- 8 new line character = 8 bytes
- total =  $64 + 64 + 8 = 136$  bytes!
- for  $n \times n$  system:  **$2n^2 + n$  bytes!**

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

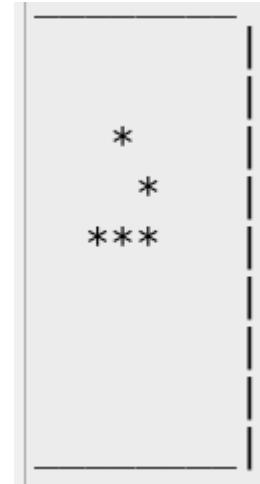
# Storing universe in Conway's game of life

## Option 2:

- get rid of spaces
- $8 \times 8 \text{ bytes} + 8 \text{ '\n'} = 72 \text{ bytes}$
- general case: **n x n +n bytes**

```
00000000
00000000
00010000
00001000
00111000
00000000
00000000
00000000
```

use blanks as dead cells, '\*' as alive



# Storing universe in Conway's game of life

## Option 3:

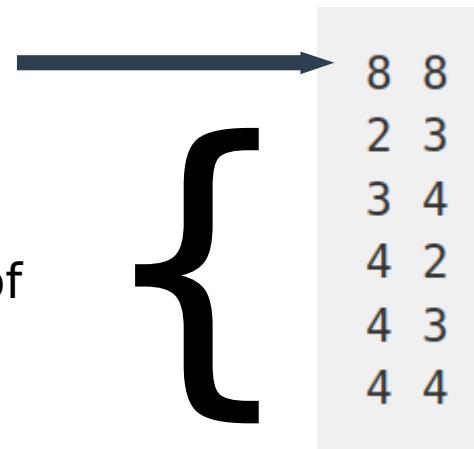
- The universe is usually sparse: relatively few live cells among many dead cells
- A consequence of the rules of the game: if an area is overcrowded, cells die
- Could we waste space by storing dead cells? (Yes!)
- So just store locations of live cells!

Storage cost for general case:  
4 bytes per live cell (per row)  
+ 4 byte (1 row) to store the  
size of the universe, hence

**4n + 4 bytes!**

Size of the universe

Coordinates of  
live cells!



It takes  $5 \times 4 + 4 = 24$  bytes  
so store this specific state!

# Storing universe in Conway's game of life

## Option 4:

Since 0's and 1's can represent dead or alive cells,

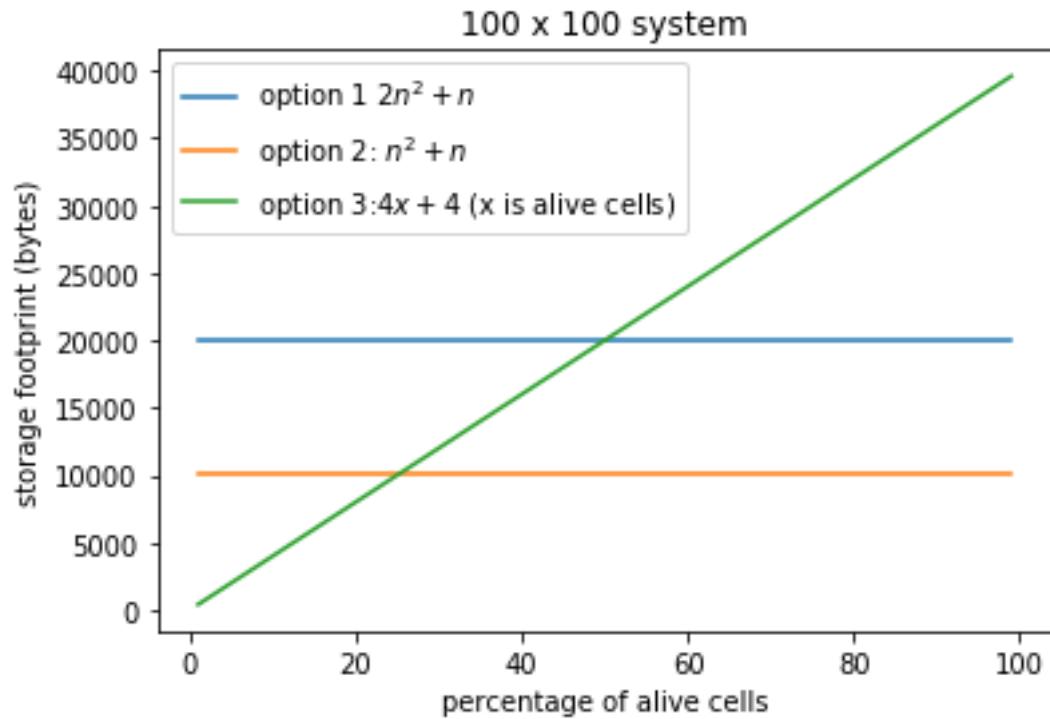
store everything in bits as opposed to bytes

Python's [binary operators](#) might enable us to do this.

However this is beyond the context of this course.

# Storing universe in Conway's game of life

## Comparing various options



for 25% or less  
alive cells,  
option 3 is superior  
to others

Note: For systems larger than 256x256,  
storage requirement per pixel will be bigger.

# Pickling

```
for coord in coords:  
    f.write(str(coord[0])+' '+str(coord[1])+'\n')
```

Recall this code?

as straightforward as it is, it seems to be a lot work

can there be a better way?

# Pickling

Enter **pickle** module:  
it preserves the object  
when writing to disk

- almost any python object can be pickled
- given so convenient, why not always use it?
- pickled representation is not human readable

```
1# life.py
2universe = [ [0, 0, 0, 0, 0, 0, 0, 0],
3              [0, 0, 0, 0, 0, 0, 0, 0],
4              [0, 0, 0, 1, 0, 0, 0, 0],
5              [0, 0, 0, 0, 1, 0, 0, 0],
6              [0, 0, 1, 1, 1, 0, 0, 0],
7              [0, 0, 0, 0, 0, 0, 0, 0],
8              [0, 0, 0, 0, 0, 0, 0, 0],
9              [0, 0, 0, 0, 0, 0, 0, 0]
10             ]
11import pickle
12f = open( 'pickled_universe.pickle' , 'wb' )
13pickle.dump(universe, f)
14f.close()
15f = open('pickled_universe.pickle', 'rb')
16u = pickle.load(f)
17f.close()
18print(u)
```

# Shelves

- Pickles can be inefficient with some data types, such as dictionaries
- Dictionary can be pickled, but in order to access a single element,
  - entire dictionary must be read into memory
  - it needs to be unpickled
  - This is a huge waste of time and space
- For this common problem, Python provides **shelve** module

A **shelve** is a dictionary that is efficiently stored in a disk

# Shelves

```
1 #shelve_example.py
2 import shelve
3 s = shelve.open('test_shelve')
4 s['bob'] = 42
5 s['liz']=[31]
6 s.close()
7
8 s = shelve.open('test_shelve')
9 for key in s.keys():
10    print(key, ':', s[key])
```

```
8 #alternative
9 with shelve.open('test_shelve') as s:
10    s['bob'] = 42
11    s['liz']=[31]
```

```
In [70]: runfile('/home/sbulut/github/cpsc128/code/python3/shelve_example.py', wdir='/
home/sbulut/github/cpsc128/code/python3')
bob : 42
liz : [31]
```

# Gotcha: Shelves update on assignment not mutation!

```
bob : 42
liz : [31]
>>> s['bob'] = 43
>>> s['liz'][0] = 30
>>> s.close()
>>> s = shelve.open('test_shelve')
>>> for key in s.keys():
print key,(':', s[key])
```

```
bob : 43
liz : [31]
```

# Gotcha: Shelves update on assignment not mutation! Solution 1

```
>>> s = shelve.open('test_shelve', writeback=True)
>>> s['liz'][0] = 1
>>> s.close()
>>> s = shelve.open('test_shelve')
>>> for key in s.keys():
print key, ':', s[key]
```

```
bob : 43
liz : [1]
```

# Gotcha: Shelves update on assignment not mutation! Solution 2

```
>>> s = shelve.open('test_shelve', writeback=True)
>>> tmp = s['liz']
>>> tmp[0] = 2
>>> s['liz'] = tmp
>>> s.close()
>>> s = shelve.open('test_shelve')
>>> for key in s.keys():
print key, '::::', s[key]
```

bob :: 43

liz :: [2]

# **Part III- Object Oriented Programming**

# Overview: Object Oriented Programming

- **Encapsulation**
- **Polymorphism**
- **Inheritance**

# Object Oriented Programming (OOP)

- **so far we've been doing:**

- procedural programming: programmer specifies a procedure, step-by-step instructions, to solve a problem
- object-based programming: we began to operate on objects

```
a_list.append('Tom')
```

- above we're invoking append method of a list object a\_list

- **objects package data and code to operate on together into a convenient unit that's easy to use**

# Object Oriented Programming (OOP)

- go beyond pre-existing classes and
- create new classes of our own
- wish Python provided `dice` and `playing_cards`?
- just add them by defining new classes
- **With this ability comes a new perspective:**
  - instead of focusing on what needs to be done (procedural)
  - focus on objects in a given problem and how they will interact
- **new jargon: encapsulation, inheritance, polymorphism**

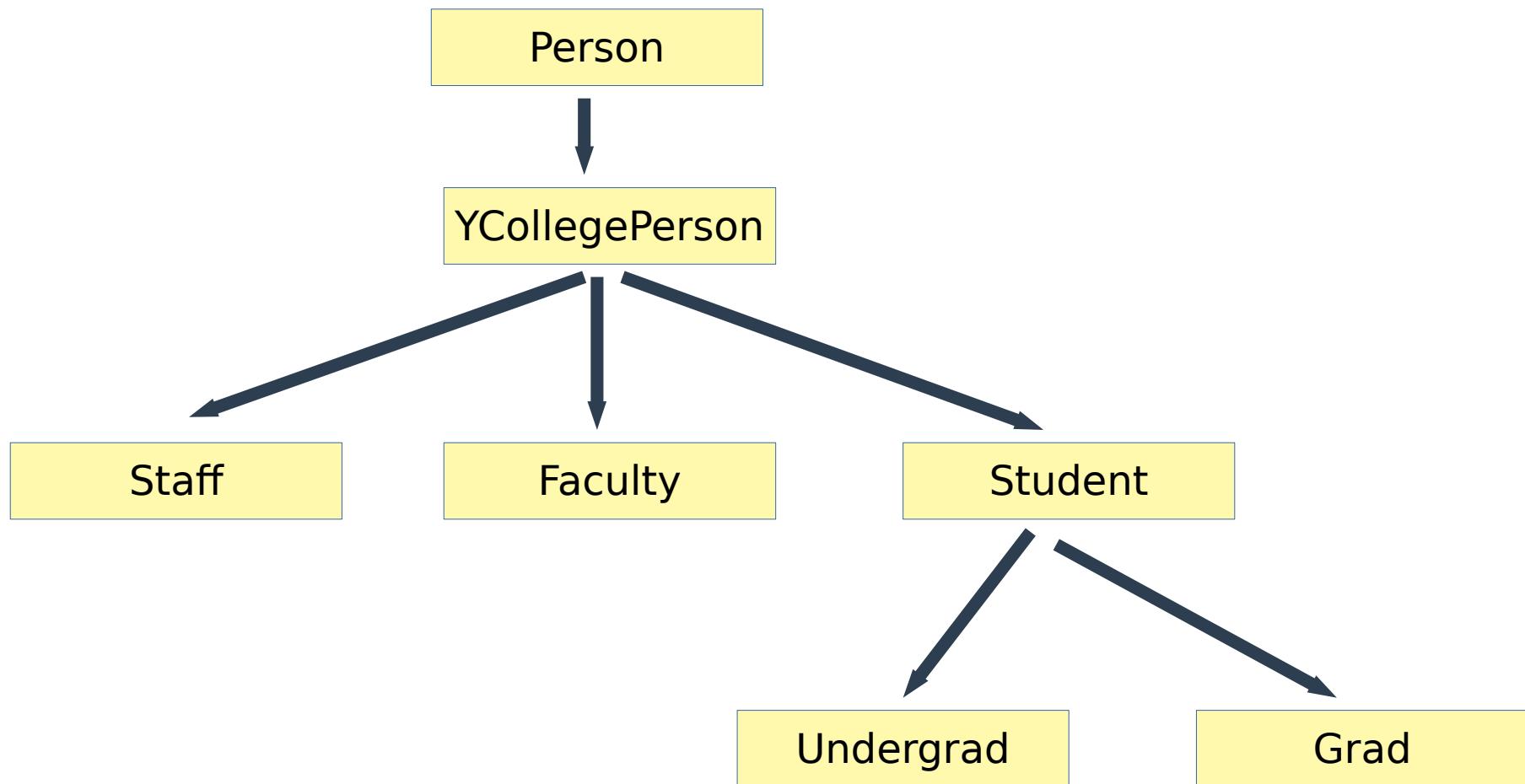
# Encapsulation

- **The first benefit of OOP is encapsulation:**
  - bundling together data attributes and methods operating on them
- **A consequence: how an object is implemented under the hood is hidden from the code invoking it**
  - also called “information hiding”
  - we used lists/dictionaries without knowing how they’re implemented
  - a program using an object only needs to know the specification of a method (input parameter etc)
- **complete encapsulation: an object type is re-written and “dropped” into a working program, without other program components needing to be aware of the change**
- **Hence: encapsulation leads to modularity & reusability**

# Inheritance

- **Experience of creating objects soon showed that**
  - many objects share similar features, hence
  - existing code can be re-used by one type object inheriting behaviour and properties from another
  - when ancestor object is updated, descendant object instantly inherits such improvements

# Inheritance: example



# Polymorphism

**An operation/request triggers different behaviour in different context**

```
x = y + 5
```

addition

```
salutation = 'Mr.' + last_name
```

append/concatenate

```
t = [3, 7] + [6, 2]
```

list expansion

# **Plan for the next three classes**

**All these OOP topics will be discussed today + in the following two lectures**

**Each lecture will be built around a main example**

- playing cards**
- Hunt the Wumps**
- Rational Numbers (Fractions)**

# A simple example: Die class

```
1# die_class_0.py
2import random
3
4class Die:
5    def __init__(self, n):
6        self.nsides = n
7
8    def roll(self):
9        spots = random.randint(1, self.nsides)
10       return spots
11
12if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end=' ')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll())
21    print('The die d1 has %d sides' % (d1.nsides))
```

# A simple example: Die class

```
1# die_class_0.py
2import random
3
4class Die:
5    def __init__(self, n):
6        self.nsides = n
7
8    def roll(self):
9        spots = random.randint(1, self.nsides)
10       return spots
11
12if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end=' ')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll())
21    print('The die d1 has %d sides' % (d1.nsides))
```

keyword **class** announces definition of a new object type

# A simple example: Die class

```
1# die_class_0.py
2import random
3
4class Die:
5    def __init__(self, n):
6        self.nsides = n
7
8    def roll(self):
9        spots = random.randint(1, self.nsides)
10       return spots
11
12if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end=' ')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll())
21    print('The die d1 has %d sides' % (d1.nsides))
```

name of this new object type is **Die**  
convention: class names are capitalized



# A simple example: Die class

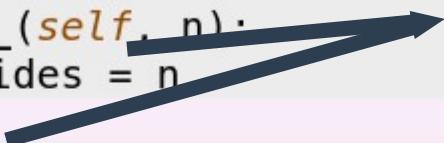
```
1# die_class_0.py
2import random
3
4class Die:
5    def __init__(self, n):
6        self.nsides = n
7
8    def roll(self):
9        spots = random.randint(1, self.nsides)
10       return spots
11
12if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end=' ')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll())
21    print('The die d1 has %d sides' % (d1.nsides))
```

**\_\_init\_\_** short for initialize  
this function is executed whenever  
an object of type **Die** is created

# A simple example: Die class

```
1# die_class_0.py
2import random
3
4class Die:
5    def __init__(self, n):
6        self.nsides = n
7
8    def roll(self):
9        spots = random.randint(1, self.nsides)
10       return spots
11
12if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end=' ')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll())
21    print('The die d1 has %d sides' % (d1.nsides))
```

All class methods should use **self** as the first parameter. It provides access to the current object



# A simple example: Die class

```
1# die_class_0.py
2import random
3
4class Die:
5    def __init__(self, n):
6        self.nsides = n
7
8    def roll(self):
9        spots = random.randint(1, self.nsides)
10       return spots
11
12if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end=' ')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll())
21    print('The die d1 has %d sides' % (d1.nsides))
```



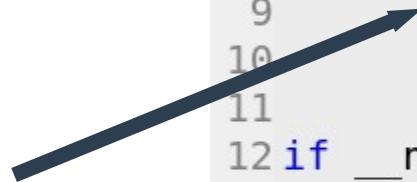
**class attribute**

Each **Die** object will have an attribute `nsides`!

# A simple example: Die class

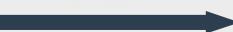
class **method**

Each **Die** object can be rolled!



```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1, self.nsides)
10        return spots
11
12 if __name__ == '__main__':
13     d1 = Die(6)
14     red = Die(20)
15
16     print('Rolling d1 ...', end=' ')
17     result = d1.roll()
18     print('result =', result)
19
20     print('Rolling red and d1 together gets y')
21     print('The die d1 has %d sides' % (d1.nsides))
```

# A simple example: Die class

```
1# die_class_0.py
2import random
3
4class Die:
5    def __init__(self, n):
6        self.nsides = n
7
8    def roll(self):
9        spots = random.randint(1, self.nsides)
10       return spots
11
12if __name__ == '__main__':
13    d1 = Die(6) 
14    red = Die(20)
15
16    print('Rolling d1 ...', end='')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll())
21    print('The die d1 has %d sides' % (d1.nsides))
```

**Object constructors:** each invokes **Die's** `__init__` method. Note: object constructor name is the same as the class name

# A simple example: Die class

```
class Die:
    5     def __init__(self, n):
        6         self.nsides = n
    7
    8     def roll(self):
        9         spots = random.randint(1, self.nsides)
    10        return spots
    11
    12if __name__ == '__main__':
    13    d1 = Die(6)
    14    red = Die(20)
    15
    16    print('Rolling d1 ...', end='')
    17    result = d1.roll() → invoke d1's roll method
    18    print('result =', result)
    19
    20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll())
    21    print('The die d1 has %d sides' % (d1.nsides))
```

access d1's attribute nsides

# Simple example: die class

```
# die_class_0.py
import random

class Die:
    def __init__(self, n):
        self.nsides = n

    def roll(self):
        spots = random.randint(1, self.nsides)
        return spots

if __name__ == '__main__':
    d1 = Die(6)
    red = Die(20)

    print 'Rolling d1 ...',
    result = d1.roll()           # Invoke d1's roll method.
    print 'result =', result

    print 'Rolling red and d1 together gets you:', d1.roll() + red.roll()
    print 'The die d1 has %d sides' % (d1.nsides)
```

The keyword `class` announces the definition of a new object type.

The name of this new object type is `Die`. By convention class names are capitalized. The class name is also used as the constructor name (see below).

`__init__` is short for initialize. This function is executed each time an object of type `Die` is created.

class attribute  
Each `Die` object will have an attribute `nsides`.

class method  
Each die object can be rolled.

All class methods should use `self` as the first parameter. It provides access to the current object.

Object constructors. Each invokes `Die`'s `__init__` method.

Access `d1`'s attribute `nsides`

# Python OOP syntax

```
class ClassName:  
    def __init__(self, ...):  
        ...  
  
    def method1(self, ...):  
        ...  
    def method2(self, ...):  
        ...
```

# A simple programming practice: Person object

- Define person class
- with attributes name & birthdate
- with method “getage”
  - use: **datetime** module
- after **if \_\_name\_\_ = '\_\_main\_\_':**
  - create two person objects:
  - p1 = Person(fullname, birthdate)
  - p2 = Person(fullname, birthdate)
  - see id(p1) not eq id(p2)
  - get their respective ages

```
In [111]: import datetime as dt  
  
In [112]: now = dt.datetime.now()  
  
In [113]: year = now.year  
  
In [114]: month = now.month  
  
In [115]: day = now.day  
  
In [116]: [year, month, day]  
Out[116]: [2019, 6, 11]
```

```
1# die_class_0.py  
2import random  
3  
4class Die:  
5    def __init__(self, n):  
6        self.nsides = n  
7  
8    def roll(self):  
9        spots = random.randint(1, self.nsides)  
10       return spots  
11
```

```
In [165]: from person import *  
  
In [166]: p1 = Person("Clark Kent", '1990/1/2')  
  
In [167]: [p1.fullname, p1.birthdate, p1.getage()]  
Out[167]: ['Clark Kent', '1990/1/2', '29 years 5 months 9days']  
  
In [168]: p2 = Person("Albert Einstein", '1890/1/2')  
  
In [169]: [id(p1), id(p2)]  
Out[169]: [140518267866808, 140518267866752]
```

# An extended example: Playing card classes

- Earlier we developed a series of functions to help write programs involving playing cards.
- Now we'll take the next step and make our code object-oriented.
- How shall we go about designing the class(es)?

There are many possible answers to this question, but for a small domain like this one (as opposed to creating an OS for example) a good approach is wish fulfillment.

- we specify the kind of code we wish we could write

# Playing card classes: The specification

We are wishing to be able to do the followings:

Get a deck of cards

Shuffle the deck

Display the deck

Create a hand of cards for roxx

Create a hand of cards for chris

Deal five cards to roxx

Display roxx's cards

Deal five cards to chris

Display chris' cards

How many cards are left in the deck?

If roxx has a flush

    congratulate him

# Playing card classes: The specification

Using OOP, the corresponding Python code will be as easy to read

```
d = Deck()
d.shuffle()
print('d after shuffling =', d)
print('d has', d.cards_left(), 'cards')
roxx = Hand()
chris = Hand()
for card in range(5):
    roxx.add(d.deal())
print('Your hand of', roxx.size(), 'cards contains:', roxx)
chris.add(d.deal(5))
print('Your hand of', chris.size(), 'cards contains:', chris)
print('There are', d.cards_left(), 'cards left in the deck.')
if roxx.is_flush():
    print('roxx rocks!')
```

This code will serve as our specification.

We'll know we're done when we've added sufficient code above this that it runs correctly.

# Identifying the necessary classes

- We can learn a lot about what code we have to write by carefully examining our specification code.
- First let's look for calls to constructors.
- A constructor uses a class name as a function.
  - We've seen examples of this with Python's built-in classes, e.g

```
>>> num = int(43.72)
>>> num
43
>>> lst = list('Tim')
>>> lst
['T', 'i', 'm']
>>>
```

# Identifying the necessary classes

- Now we want to look for similar syntax in our specification code,

```
d = Deck()  
d.shuffle()  
print('d after shuffling =', d)  
print('d has', d.cards_left(), 'cards')  
roxx = Hand()  
chris = Hand()  
for card in range(5):  
    roxx.add(d.deal())  
print('Your hand of', roxx.size(), 'cards contains:', roxx)  
chris.add(d.deal(5))  
print('Your hand of', chris.size(), 'cards contains:', chris)  
print('There are', d.cards_left(), 'cards left in the deck.')  
if roxx.is_flush():  
    print('roxx rocks!')
```

Three calls to two different constructors means our code requires two classes for sure:  
**Hand** and **Deck**

# Identifying the classes' methods

```
d = Deck()
d.shuffle()
print('d after shuffling =', d)
print('d has', d.cards_left(), 'cards')
roxx = Hand()
chris = Hand()
for card in range(5):
    roxx.add(d.deal())
print('Your hand of', roxx.size(), 'cards contains:', roxx)
chris.add(d.deal(5))
print('Your hand of', chris.size(), 'cards contains:', chris)
print('There are', d.cards_left(), 'cards left in the deck.')
if roxx.is_flush():
    print('roxx rocks!')
```

**Deck** class must provide:

- shuffle()
- cards\_left()
- deal()
- deal() should default to 1 card  
if no parameter is given

**Hand** class should provide

- add()
- size()
- is\_flush() etc

# Program skeleton 1

Notes:

We're using **pass** statement as placeholder

**\_\_init\_\_** is required for each class:  
it will contain code necessary to construct objects of the type specified by the class

```
1 # playing_cards_skeleton_1.py
2 class Deck:
3     def __init__(self):
4         pass
5     def shuffle(self):
6         pass
7     def cards_left(self):
8         pass
9     def deal(self, n=1):
10        pass
11
12 class Hand:
13     def __init__(self):
14         pass
15     def add(self, cards):
16         pass
17     def size(self):
18         pass
19     def is_flush(self):
20         pass
21
22 d = Deck()
23 d.shuffle()
24 print('d after shuffling =', d)
25 print('d has', d.cards_left(), 'cards')
26 roxx = Hand()
27 chris = Hand()
28 for card in range(5):
29     roxx.add(d.deal())
30 print('Your hand of', roxx.size(), 'cards contains:', roxx)
31 chris.add(d.deal(5))
32 print('Your hand of', chris.size(), 'cards contains:', chris)
33 print('There are', d.cards_left(), 'cards left in the deck.')
34 if roxx.is_flush():
35     print('roxx rocks!')
```

# An invisible method

```
In [102]: runfile('/home/sbulut/github/cpsc128/code/python3/  
playing_cards_skeleton1.py', wdir='/home/sbulut/github/cpsc128/code/python3')  
d after shuffling = <__main__.Deck object at 0x7f94e0442400>  
d has None cards  
Your hand of None cards contains: <__main__.Hand object at 0x7f94e0442860>  
Your hand of None cards contains: <__main__.Hand object at 0x7f94e0442a90>  
There are None cards left in the deck.
```

```
In [103]:
```

# Program Skeleton 2

Note:

we can't use **pass** as placeholder  
in `__str__`  
since the method is required to  
return a string

```
In [103]: runfile('/home/sbulut/github/cps  
playing_cards_skeleton2.py', wdir='/home/s  
d after shuffling =  
d has None cards  
Your hand of None cards contains:  
Your hand of None cards contains:  
There are None cards left in the deck.  
  
In [104]:
```

```
1 # playing_cards_skeleton_2.py  
2 class Deck:  
3     def __init__(self):  
4         pass  
5     def shuffle(self):  
6         pass  
7     def cards_left(self):  
8         pass  
9     def deal(self, n=1):  
10        pass  
11    def __str__(self):  
12        return ''  
13  
14 class Hand:  
15     def __init__(self):  
16         pass  
17     def add(self, cards):  
18         pass  
19     def size(self):  
20         pass  
21     def is_flush(self):  
22         pass  
23     def __str__(self):  
24        return ''  
25  
26 d = Deck()  
27 d.shuffle()  
28 print('d after shuffling =', d)  
29 print('d has', d.cards_left(), 'cards')  
30 roxx = Hand()  
31 chris = Hand()  
32 for card in range(5):  
33     roxx.add(d.deal())  
34 print('Your hand of', roxx.size(), 'cards contains:', roxx)  
35 chris.add(d.deal(5))  
36 print('Your hand of', chris.size(), 'cards contains:', chris)  
37 print('There are', d.cards_left(), 'cards left in the deck.')  
38 if roxx.is_flush():  
39     print('roxx rocks!')
```

# A new class: Card

- **as far as our specs go, we have done well so far**
- **but have to realize that Deck and Hand are both a collection of cards**
- **instead of both classes having duplicate code for manipulating and/or displaying cards**
- **we can create a new object/class called “Card”**

# Program Skeleton 3



```
1 # playing_cards_skeleton_3.py
2 class Card:
3     pass #not sure what to put here yet
4
5 class Deck:
6     def __init__(self):
7         self.cards = []
8         for cardnum in range(52):
9             self.cards.append(Card(cardnum))
10    def shuffle(self):
11        pass
12    def cards_left(self):
13        pass
14    def deal(self, n=1):
15        pass
16    def __str__(self):
17        return '' #can't use pass here
18
19 class Hand:
20    def __init__(self):
21        pass
22    def add(self, cards):
23        pass
24    def size(self):
25        pass
26    def is_flush(self):
27        pass
28    def __str__(self):
29        return '' #can't use pass here
30
31 d = Deck()
32 d.shuffle()
33 print('d after shuffling =', d)
34 print('d has', d.cards_left(), 'cards')
35 roxx = Hand()
36 chris = Hand()
37 for card in range(5):
38     roxx.add(d.deal())
39 print('Your hand of', roxx.size(), 'cards contains:', roxx)
40 chris.add(d.deal(5))
41 print('Your hand of', chris.size(), 'cards contains:', chris)
42 print('There are', d.cards_left(), 'cards left in the deck.')
43 if roxx.is_flush():
44     print('roxx rocks!')
45
```

# Deck methods

```
20 class Deck:
21     def __init__(self):
22         self.cards = []
23         for cardnum in range(52):
24             self.cards.append(Card(cardnum))
25
26     def shuffle(self):
27         #Knuth's swap shuffle: random swap from ordered part to shuffled one
28         ncards = len(self.cards)
29         for swaps in range(ncards):
30             #loop from ncards-1 to 0 instead of 0 to ncards-1
31             swaps = ncards - 1 - swaps
32             posn1 = random.randint(0, swaps)
33             self.cards[posn1], self.cards[swaps] = self.cards[swaps], self.cards[posn1]
34
35     def cards_left(self):
36         return len(self.cards)
37
38     def deal(self, n=1):
39         c = self.cards[-n:]
40         del(self.cards[-n:])
41         return c
42
43     def __str__(self):
44         s = ''
45         for card in self.cards:
46             s = s + str(card) + ', '
47         return s
```

- A natural choice for Deck is a list  
- Initialize a deck with 52 cards

# Hand Methods

```
49 class Hand:
50     def __init__(self):
51         self.cards = []
52
53     def add(self, cards):
54         self.cards.extend(cards)
55
56     def size(self):
57         return len(self.cards)
58
59     def is_flush(self):
60         pass
61
62     def __str__(self):
63         s = ''
64         for card in self.cards:
65             s = s + str(card) + ', '
66
67         return s
```

# Card methods

```
4 class Card:
5     FACE_VALUES = ['A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K']
6     SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
7
8     def __init__(self, cardnum):
9         self.number = cardnum
10
11    def __str__(self):
12        return self.face_value() + ' of ' + self.suit()
13
14    def face_value(self):
15        return Card.FACE_VALUES[self.number % 13]
16
17    def suit(self):
18        return Card.SUITS[self.number / 13]
19
```

# Putting it all together [demo]

```
1 # playing_cards_4_v2.py
2 import random
3
4 class Card:
5     FACE_VALUES = ['A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K']
6     SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
7
8     def __init__(self, cardnum):
9         self.number = cardnum
10
11    def __str__(self):
12        return self.face_value() + ' of ' + self.suit()
13
14    def face_value(self):
15        return Card.FACE_VALUES[self.number % 13]
16
17    def suit(self):
18        return Card.SUITS[self.number // 13]
19
20 class Deck:
21     def __init__(self):
22         self.cards = []
23         for cardnum in range(52):
24             self.cards.append(Card(cardnum))
25
26     def shuffle(self):
27         #Knuth's swap shuffle: random swap from ordered part to shuffled one
28         ncards = len(self.cards)
29         for swaps in range(ncards):
30             #loop from ncards-1 to 0 instead of 0 to ncards-1
31             swaps = ncards - 1 - swaps
32             posn1 = random.randint(0, swaps)
33             self.cards[posn1], self.cards[swaps] = self.cards[swaps], self.cards[posn1]
34
35     def cards_left(self):
36         return len(self.cards)
37
38     def deal(self, n=1):
39         c = self.cards[-n:]
40         del(self.cards[-n:])
41         return c
42
43     def __str__(self):
44         s = ''
45         for card in self.cards:
46             s = s + str(card) + ', '
47         return s
48
```

```
49 class Hand:
50     def __init__(self):
51         self.cards = []
52
53     def add(self, cards):
54         self.cards.extend(cards)
55
56     def size(self):
57         return len(self.cards)
58
59     def is_flush(self):
60         pass
61
62     def __str__(self):
63         s = ''
64         for card in self.cards:
65             s = s + str(card) + ', '
66         return s
67
68 if __name__ == '__main__':
69     d = Deck()
70     d.shuffle()
71     print('d after shuffling =', d, '\n')
72     print('d has', d.cards_left(), 'cards.\n')
73
74     roxx = Hand()
75     chris = Hand()
76
77     for card in range(5):
78         roxx.add(d.deal())
79     print('Your hand of', roxx.size(), 'cards contains:', roxx)
80
81     chris.add(d.deal(5))
82     print('Your hand of', chris.size(), 'cards contains:', chris)
83
84     print('\nThere are', d.cards_left(), 'cards left in the deck.')
85     if roxx.is_flush():
86         print('roxx rocks!')
```

# Running the whole thing

```
In [3]: runfile('/home/sbulut/github/cpsc128/code/python3/playing_cards_4_ttoper_v2.py', wdir='/home/sbulut/github/cpsc128/code/python3')
d after shuffling = 4 of Clubs, 3 of Spades, 2 of Hearts, 2 of Spades, 5 of Hearts, 7 of Diamonds, 9 of Spades, T of Diamonds, 6 of Spades, K of Clubs, 7 of Hearts, T of Hearts, T of Spades, K of Hearts, 2 of Diamonds, 5 of Clubs, 8 of Hearts, 3 of Diamonds, J of Hearts, J of Spades, 8 of Clubs, 3 of Hearts, 8 of Spades, 6 of Clubs, Q of Clubs, 6 of Diamonds, J of Diamonds, 7 of Clubs, 2 of Clubs, Q of Diamonds, 9 of Diamonds, 3 of Clubs, 5 of Diamonds, A of Clubs, 4 of Spades, 8 of Diamonds, 6 of Hearts, Q of Spades, 7 of Spades, A of Hearts, K of Diamonds, 9 of Clubs, A of Spades, T of Clubs, K of Spades, 4 of Hearts, J of Clubs, 9 of Hearts, 5 of Spades, 4 of Diamonds, A of Diamonds, Q of Hearts,
```

d has 52 cards.

Your hand of 5 cards contains: Q of Hearts, A of Diamonds, 4 of Diamonds, 5 of Spades, 9 of Hearts,  
Your hand of 5 cards contains: A of Spades, T of Clubs, K of Spades, 4 of Hearts, J of Clubs,

There are 42 cards left in the deck.

In [4]:

# Programming observation

**Note how we worked backwards**

- from the desired code use
- to required classes

**As we started implementing the specs,  
we've realized the limitations of specs and new  
requirements**

# **June 13: Object Oriented Design**

# Object Oriented Design (OOD)

- **in the previous class we focused on**
  - OOP syntax
  - programming style to write object oriented code
- **today we will focus on how to design OO code**
  - analyze problem domain into a set of objects, classify obj. typ
  - write a class to represent each object
  - instantiate classes to create individual objects
  - write the 'main' program, specifying communication/interaction between objects

# OOD: Analysis

- **critical step: divide problem domain into right set of objects**
- **in playing cards, this was fairly obvious, and there was little interaction/communication between objects**
  - typically this is not the case
  - in non-trivial problems equally plausible decomposition of problem into a different set of objects is possible!
- **easiest case: problem domain consists of physical objects (packages, boxes, trucks, warehouses)**
- **hardest cases:**
  - no physical objects, or objects do not leave physical traces
  - too many physical objects (climate modelling & molecules), requiring some degree of **abstraction**

# OOD: Classes

- once types of objects identified, we can start writing classes
- we need to identify objects' attributes:
  - What **qualities** does this object have? weight, color, size, owner, position
  - What **information** can it **store**?
- we need to identify objects' methods:
  - What can it **do**? what kind of requests can it respond to? Can it move, cancel, record, trigger?
  - What questions can it **answer** about itself? size, position, colour?
- **attributes correspond to adjectives & nouns**
- **methods correspond to verbs**

# OOD: Instantiation & main program

## - instantiation

- creating objects the program requires, consisting of calls to the object constructors

```
d = Deck()  
...  
roxx = Hand()  
chris = Hand()
```

## - writing the main program

- the last step to complete the program
- specifies the pattern of interaction between objects in the system

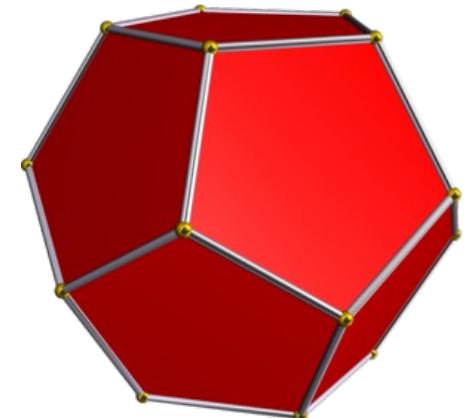
# **Extended Example: Hunt the Wumpus!**

- **in playing card example,**
  - objects were fairly obvious
  - not much communication/interaction between objects
  - this is not typical!
- **in this extended example we will consider**
  - problem with more objects
  - less obvious analysis than playing card
  - with considerably more interaction/communication b/w objcts
- **our task is to implement the game:  
Hunt the Wumpus!**

# About “Hunt the Wumpus”

- a text based survival horror type adventure game
- developed in 1973 by Gregory Yob
- player moves through a series of connected caves (arranged in dodecahedron)
- and hunts a monster named Wumpus
- while avoiding fatal bottomless pits & super bats
- player fires a ‘crooked arrow’ to kill Wumpus
- originally written in BASIC

```
You are in room 9.  
Tunnels lead to 2, 4, 12.  
Shoot or Move (S-M)? M  
Where to? 12  
  
You are in room 12.  
I smell a Wumpus.  
Tunnels lead to 3, 11, 19.  
Shoot or Move (S-M)? S  
No. of Rooms (1-5)? 1  
Room # 13  
RAAA! You got the wumpus!  
HEE HEE HEE - The Wumpus'll get you next time!!  
Same setup (Y-N)? Y  
  
You are in room 2.  
I feel a draft.  
Tunnels lead to 1, 3, 10.  
Shoot or Move (S-M)? M  
Where to? 3  
  
You are in room 3.  
Tunnels lead to 2, 4, 12.  
Shoot or Move (S-M)? M
```



# Hunt the Wumpus [Demo]

Check out:

<https://jayisgames.com/games/hunt-the-wampus/>

**Hunt the Wumpus**

 add to favorites  55k

 4.1/5 (171 votes)

HAS 3 TUNNELS LEADING TO OTHER ROOMS. (LOOK AT A DODECAHEDRON TO SEE HOW THIS WORKS-IF YOU DON'T KNOW WHAT A DODECAHEDRON IS, ASK SOMEONE)

HAZARDS:  
BOTTOMLESS PITS - TWO ROOMS HAVE BOTTOMLESS PITS IN THEM  
IF YOU GO THERE, YOU FALL INTO THE PIT (& LOSE!)  
SUPER BATS - TWO OTHER ROOMS HAVE SUPER BATS. IF YOU  
GO THERE, A BAT GRABS YOU AND TAKES YOU TO SOME OTHER  
ROOM AT RANDOM. (WHICH MAY BE TROUBLESOME)

WUMPUS:  
THE WUMPUS IS NOT BOthered BY HAZARDS (HE HAS SUCKER  
FEET AND IS TOO BIG FOR A BAT TO LIFT). USUALLY  
HE IS ASLEEP. TWO THINGS WAKE HIM UP: YOU SHOOTING AN

# OO Analysis

- we have a pretty good informal specification of the problem:
  - description of the game and
  - its rules are displayed on start
- remembering that objects correspond to nouns,
- read over description and highlight objects

# OO Analysis

Welcome to "Hunt the Wumpus"

The wumpus lives in a cave of 20 rooms . Each room has 3 tunnels to other rooms. (Look at a dodecahedron to see how this works. If you don't know what a dodecahedron is, ask someone.)

Hazards:

Bottomless pits - Two rooms have bottomless pits in them. If you go there, you fall into the pit (& lose)!

Super bats - Two other rooms have super bats. If you go there, a bat grabs you and takes you to some other room at random (which may be troublesome).

Wumpus:

The wumpus is not bothered by hazards. (He has sucker feet and is Too big for a bat to lift.) Usually he is asleep. Two things wake him up: Your shooting an arrow , or your entering his room. If the wumpus wakes, he moves ( $p=.75$ ) one room or stays still ( $p=.25$ ). After that, if he is where you are, he eats you up and you lose!

# OO Analysis

You:

Each turn you may move or shoot a crooked arrow.

Moving: You can move one room (Through one tunnel).

Arrows: You have 5 arrows. You lose when you run out. Each arrow can go from 1 to 3 rooms. You aim by telling the computer the rooms to which you want the arrow to go. If the arrow can't go that way (if no tunnel) it moves at random to the next room.

If the arrow hits the wumpus, you win.

If the arrow hits you, you lose.

Warnings:

When you are one room away from a wumpus or hazard, the computer says:

Wumpus: "I smell a wumpus!"

Bat : "Bats nearby!"

Pit : "I feel a draft!"

You are in room 12.

There are tunnels to rooms 11, 13, and 19.

I feel a draft!

Shoot, move, or quit (s/m/q)? m

To which room (11, 13, or 19)? 13

AEEEEIIIII! You fell into a pit!

# OO Analysis

Welcome to "Hunt the Wumpus"

The wumpus lives in a cave of 20 rooms. Each room has 3 tunnels to other rooms. (Look at a dodecahedron to see how this works. If you don't know what a dodecahedron is, ask someone.)

Hazards:

Bottomless pits - Two rooms have bottomless pits in them. If you go there, you fall into the pit (& lose)!

Super bats - Two other rooms have super bats. If you go there, a bat grabs you and takes you to some other room at random (which may be troublesome).

Wumpus:

The wumpus is not bothered by hazards. (He has sucker feet and is too big for a bat to lift.) Usually he is asleep. Two things wake him up: Your shooting an arrow, or your entering his room.

If the wumpus wakes, he moves (n= 75) one room or stays still.

# OO Analysis

The types of objects we have then are:

- a wumpus;
- a cave system of
- rooms connected by
- tunnels;
- pits;
- you, i.e. the player,
- bats; and
- arrows.

From the rules, we can tell how objects will behave:

- Wumpus mostly sleeps, wakes when disturbed, and reacts in a random way
- Bats grab players and fly them somewhere else
- You, the player, move from room to room, and shoot arrows etc.

# Object attributes and methods

Keeping the objects we've identified in mind we can re-read the game description looking for verbs and adjectives to reveal the objects' attributes and methods.

## Player

- Attributes: At any instant the player has a **location** in the cave system and a **number of arrows** left.
- Methods: The player is able to **move** from room to room and to **shoot** arrows.

## Bat

- Attributes: Like the player(s) each bat has a **location** in the cave system.
- Methods: A Bat's only action is to **snatch** the player and carry him or her away.

## Pit

- Attributes: Like Players and Bats Pits have a **location**.
- Methods: Pits can **swallow** players who stumble into them.

## Wumpus

- Attributes: **Location**
- Methods: All the wumpus does is **wake up** and either run off or eat a player.

## Arrows

- Attributes: A **path** they will try and follow.
- Methods: An Arrow **flies** from room to room along the specified path.

# Cave system objects

The cave system is the backbone of this system of objects:

- all the simple objects are situated in and/or move through it.

It is the most complex of the objects because it is a compound object:

- the cave system consists of multiple rooms connected by multiple tunnels.
- it will have to be a data container type of some sort.

Attributes: 20 **caves**; the **pattern of connections** between caves.

Methods: Should be able to tell us **where you can go from here** (i.e. what rooms are connected to this one); if a room **has a bat**; if a room **has a pit**; if the **player is in a room**.

The cave system contains 20 Room objects which as objects have their own attributes and methods:

Attributes: 3 **tunnels**, possibly **a bat**, possibly **a pit**, possibly **a wumpus**.

Methods: Should be able to tell us if it **has a bat**, **has a pit**, or **has a wumpus**.

# Main routine Pseudocode

Let's use these class, attribute and method names we've identified so far and try to write the main routine of our program.

Doing so should reveal any missing objects, or missing attributes or methods of objects.

First, we might want to write pseudocode as plainly as we can in English.

```
While the game isn't over
    Display the game state
    Display the menu of possible actions
    Get the user's choice of action
    If the action is move
        Display the choices (connected rooms)
        Get the user's choice
        Move the player
        If the room has a bat
            Have the bat snatch the player
        Otherwise if the room has a pit
            Have the pit swallow the player
        Otherwise if the room has a Wumpus
            Wake the Wumpus up
        Otherwise if the action is shoot
            Get the path for the arrow to follow from the user
            Tell the arrow to fly that path
        Otherwise if the action is quit
            Quit the game
```

# Main routine Pseudocode

```
1 while not game_over:  
2     print cave_system  
3     print player.location  
4     action = input('Shoot, move or quit (s/m/q)? ')  
5     if action == 'm':  
6         print 'Choose from: ', cave_system.rooms[player.location].tunnels  
7         room_choice = int(input('Your choice? '))  
8         player.move(room_choice)  
9         new_room = cave_system.rooms[player.location] # Note simplifying alias.  
10        if new_room.has_bat():  
11            new_room.bat.snatch()  
12        elif new_room.has_pit():  
13            new_room.pit.swallow()  
14            game_over = True  
15        elif new_room.has_wumpus():  
16            game_over = room.wumpus.wake_up()  
17        elif action == 's':  
18            player.shoot()  
19        elif action == 'q':  
20            game_over = True
```

# class Cave\_System

- In writing classes, one usually goes from simplest to most complicated
- in this case we will start the Cave\_System, the most complicated one, as all other objects refer to it
- Remember our initial specs for this class:
  - **Attributes:** 20 caves; the pattern of connections between caves.
- Looking at our pseudocode, it seems we picked a good “container” for this info:

```
6     print('Choose from: ', cave_system.rooms[player.location].tunnels )
...
9     new_room = cave_system.rooms[player.location] # Note simplifying alias.
```

- This shows: , **cave\_system** has **rooms** attribute, which is a list
- **rooms** is a list of **Room** objects

How can represent pattern of connections between Rooms?

- One simple way is for each **Room** to know to what other **Rooms** it is connected
- it is reasonable for a **Room** to know where it is connected to
- That is in fact what **tunnels** attribute yields (possibly a list of rooms)

# class Cave\_System

- what about initial specs for methods:

Methods: Should be able to tell us where you can go from here/what rooms are connected to this one; if a room has a bat; if a room has a pit; if the player is in a room.

- if we look at our pseudocode, we see that these questions are asked of Room objects not cave\_system

- these methods belong

to Rooms

- So, does cave\_system have any method?

- Looking at line 2, we realize a \_\_str\_\_ method is needed!

```
10      if new_room.has_bat():
...
12      elif new_room.has_pit():
...
15      elif new_room.has_wumpus():
...
```

```
2      print cave_system
```

# class cave\_system

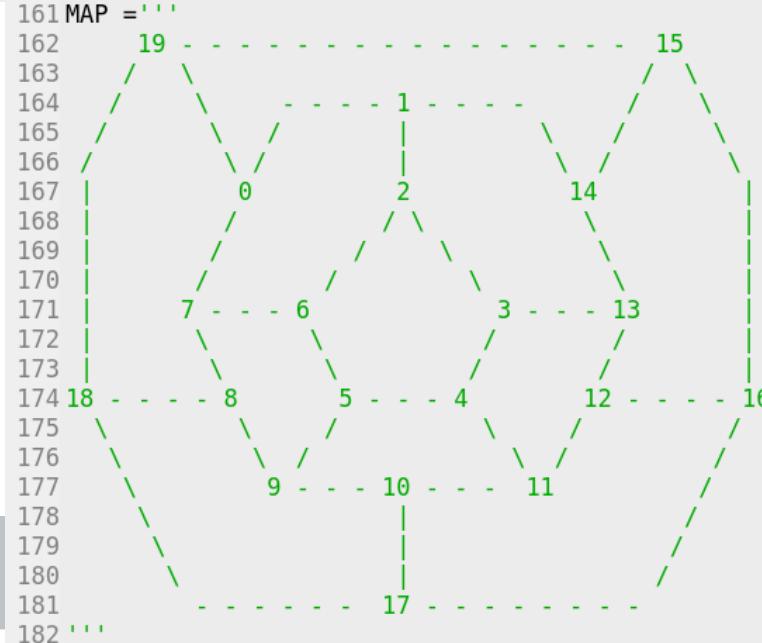
In summary, `cave_system` has only two methods: `__init__` and `__str__`.

```
20 class Cave_System:  
21  
22     def __init__(self, tunnels, system_map):  
23         self.rooms = []  
24         for i in range(len(tunnels)):  
25             self.rooms.append( Room(i, tunnels[i]) )  
26  
27         self.map = system_map  
28  
29     def __str__(self):  
30         return self.map
```

```
155 TUNNELS = [[1,7,19], [0,2,14], [1,3,6], [2,4,13], [3,5,11],  
156     [4,6,9], [2,5,7], [0,6,8], [7,9,18], [5,8,10],  
157     [9,11,17], [4,10,12], [11,13,16], [3,12,14], [1,13,15],  
158     [14,16,19], [12,15,17], [10,16,18], [8,17,19], [0,15,18]  
159 ]
```

`__str__` **has** to return a string, so we define a system map using chars

To the left we see a squashed dodecahedron



# class Room

- Rooms have the attributes and methods we identified in our first pass **plus** the methods we have shifted from Cave\_Systems to Rooms
  - Attributes: 3 **tunnels**, possibly a **bat**, possibly a **pit**, possibly a **wumpus**.
  - Methods: Should be able to tell us if it **has a bat**, **has a pit**, or **has a wumpus**,
- PLUS
  - "should be able to tell us **where you can go from here**/what rooms are connected to this one; if a room **has a bat**; if a room **has a pit**; if the **player is in a room**.
- As a result, we got rid of duplicate methods **has坑**, **has\_wumpus**, etc any other duplicate stuff?  
“player is in room?” is duplicate, as can be seen in pseudocode where we do: **player.location**, i.e, we ask the player not the room if s/he is in it.
- So remove this duplicated method from **Room** class

# class Room

## Notes:

- We will identify rooms by their number as shown in MAP previously.
- connections here is a list of rooms connected by tunnels to this one, e.g. [1, 7, 4] meaning this room is connected to room number 1, room number 7, and room number 4.
- By default a room does not have a bat, pit or wumpus (b=None, p=None, w=None). These will be sprinkled around the cave system later.
- The has\_... methods that answer questions look to see if the relevant attribute is defined and if it is returns True, otherwise False.

```
32 class Room:  
33     '''e.g. r = Room(0,[1,7,4])'''  
34     def __init__(self, room_number, connections, \  
35                 b = None, p = None, w = None):  
36         self.number = room_number  
37         self.tunnels = connections  
38         self.bat = b  
39         self.pit = p  
40         self.wumpus = w  
41  
42     def has_bat(self):  
43         if self.bat:  
44             return True  
45         else:  
46             return False  
47  
48     def has_pit(self):  
49         if self.pit:  
50             return True  
51         else:  
52             return False  
53  
54     def has_wumpus(self):  
55         if self.wumpus:  
56             return True  
57         else:  
58             return False  
--
```

# class Pit

- recall that **Pit** has one attribute: **location** and one method: **swallow**

```
78 class Pit:  
79     def __init__(self, room_number):  
80         self.location = room_number  
81  
82     def swallow(self):  
83         # What a pit does when it swallows a player.  
84         print('AIEEEE! You\'re falling to your doom down the pit!')  
85
```

In a modern graphic game a Pit's swallow method would trigger some fabulous animation of the player tumbling downward, but since this is text-based we have to settle for saying what is happening.

# class Bat

- Like **Pits**, **Bats** have one attribute, **location**, and one method called **snatch**.
- Snatching is a little more involved than swallowing.
- The bat has to move the player to a new randomly chosen room and
- then flap off to another randomly chosen room.

```
29 #with bug
30 class Bat:
31     def __init__(self, roomnumber):
32         self.location = roomnumber
33
34     def snatch(self, player):
35         print()
36         print('Aaah! A giant bat has snatched you!')
37         print()
38         self.location = random.randint(0, len(cave_system.rooms) - 1) # BUG!
39         player.location = random.randint(0, len(cave_system.rooms) - 1)
40         print("I am ", self, "in room number", self.location)
41
```

Above, can you determine what the bug is ?

# class Wumpus

Like **Bats**, **Wumpus** has a **location** attribute and a single method called **wake\_up**

```
86 class Wumpus:
87     def __init__(self, roomnumber):
88         self.location = roomnumber
89
90     def wake_up(self):
91         ''' Wake Wumpus up and see what happens.
92             returns True if the Wumpus eats the player,
93             and False otherwise.'''
94         # 1 in 4 chance Wumpus eats the player,
95         if random.randint(1,4) == 4:
96             print('Uh oh the Wumpus got you!')
97             return True
98         # and 3 in 4 chance it runs to an adjacent room.
99         else:
100             print('Wheww! The Wumpus woke up but he ran away.')
101             self.location = random.choice(cave_system.rooms[self.location].tunnels) #
102             return False
103
```

# class Player

- Like the other objects in the cave system **Players** have **locations**.
- They also have two methods: they can **move** and they can **shoot** arrows.

```
103
104 class Player:
105     def __init__(self, roomnumber=0):
106         self.location = roomnumber
107
108     def move(self, new_location):
109         if new_location in cave_system.rooms[self.location].tunnels:
110             self.location = new_location
111         else:
112             print("You can't walk through walls in this game!")
113
114     def shoot(self):
115         # You will be filling this method in!
116         print("Shooting ...")
117
```

Note that the move method receives an argument saying what room to move to, but it doesn't blindly trust the argument to be valid and instead checks that the destination room is linked to the current room.

You will be filling in the **shoot** method!

# Putting the pieces together

Spend some time on analyzing **Wumpus\_4.py**

This is the longest program in the course so take some time to read it over. You'll notice that there is:

- some new code (whose details to be explained shortly),
- some bugs (that you'll be correcting as part of the assignment), and
- a few pieces that are missing (that you'll be adding as part of the assignment).

# New code: Initializing the bats

- see `wumpus_4.py`

- Notes:

1. number of bats is not ‘hardcoded’:

instead we defined a global constant/parameter

`NUMBATS`

The jargon: the number of bats has been parameterized!

2. note `random.choice`: it chooses an item at random from a list

```
room = random.choice(cave_system.rooms)
#vs
room = cave_system.rooms[random.randint(0, len(cave_system.rooms) - 1)]
```

# New code (pointing to a missing class): shoot and Arrows

# New code: `show_cheats()`

**see `wumpus_4.py`**

# New code: Display status and warnings

**-see wumpus\_4.py**

# Bug: Fixing snatch

- Assume **Bat** is in Room 12, and goes to Room 8 after a snatch
- Then we do:  
`self.location = random.randint(0, len(cave_system.rooms) - 1)`
- Although **Bat**'s location is updated
  - Room 12 doesn't know that it doesn't have a **Bat** anymore
  - Room 8 doesn't know that it has a **Bat**
- so, in addition to **Bat**'s location to Room 8, we need to do:
  - set Room 12's bat attribute to None
  - set Room 8's bat attribute to reference this **Bat** object

## Another small Bug

When the bat's new room is chosen

- it shouldn't be a room with a Bat already in it,
- nor should it be the room it has just dropped the player in (which will happen approximately 1 in 20 times since we are choosing randomly).
- Fixing this secondary bug will be part of your assignment.

# Two more (small) problems

Bug: **Wumpus** movement

The wumpus position is not updated properly after moving for the same reason that **Bats**' movements were not properly updating. Fixing this will be part of your assignment.

Missing Piece: Initialize Pits

You have probably noticed that although the **Bats** and **Wumpus** are initialized, the Pits are not. Adding this is another part of your assignment.

# **June 18: Polymorphism & Inheritance**

# Inheritance & Polymorphism

- **three key concepts in OOP are:**
  - Encapsulation
  - inheritance
  - polymorphism
- **so far we've focused on encapsulation**
- **inheritance enables one class to inherit properties (attributes, methods) of another one**
- **polymorphism: ability of same operation to trigger different behaviours in different contexts**

# Inheritance example: Playing Cards

- Recall Deck and Hand playing card classes
- Deck and Hand are obviously different
- What about similarities:

- both are collections of the same type of object: playing cards
- both need to be displayed
- at times we need to remove cards from both of them
- at other times we need to add cards to them
- Finally, we query how many cards are in them

- OOP recommends:

**(i) removing common functionality from both Deck and Hand to a seperate base class,**

**(ii) and Deck and Hand inherit functionality from it**

**- a consequence: when improvements made to the base class, they will automatically propagate to the descendants, Deck and Hand, which is efficient**

# class CardCollection

```
class CardCollection:  
    def __init__(self):  
        self.cards = []  
  
    def size(self):  
        return len(self.cards)  
  
    def add(self, card):  
        self.cards.append(card)  
  
    def remove(self):  
        return self.cards.pop()  
  
    def __str__(self):  
        return ', '.join( str(card) for card in self.cards )
```

everything here should be familiar: see previous version of Deck and Hand classes

# class Deck

This example shows quite a few things we can do with inheritance

```
1 class Deck(CardCollection):
2     # Override ancestor's constructor, i.e. replace the default.
3     def __init__(self):
4         self.cards = []
5         for cardnum in range(52):
6             self.add( Card(cardnum) )
7
8     # Alias the inherited method "size" as "cards_left",
9     # because we usually ask how many cards are left in a
10    # deck rather than asking about its size.
11    def cards_left(self):
12        return self.size()
13
14    # Another alias. When using a deck of cards we talk about "dealing"
15    # cards not "removing" them from the deck.
16    def deal(self):
17        return self.remove()
18
19    # Add a new method, shuffle, that does not exist in ancestor class.
20    def shuffle(self):
21        ncards = len(self.cards)
22        for swaps in range(ncards):
23            swaps = ncards - 1 - swaps
24            posn1 = random.randint(0, swaps)
25            self.cards[posn1], self.cards[swaps] = self.cards[swaps], self.cards[posn1]
```

# class Hand

The code for class Hand may surprise you:

```
class Hand(CardCollection):  
    pass
```

**Hand** objects:

- don't have any extra methods that **CardCollections** do not
- nor do they customize any **CardCollection** methods

Effectively with the code above, **Hand** is an alias for **CardCollection**

# putting the pieces together

```
1 # playing_cards_5.py
2 import random
3
4 class Card:
5     FACE_VALUES = ['A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K']
6     SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
7
8     def __init__(self, cardnum):
9         self.number = cardnum
10
11    def __str__(self):
12        return self.face_value() + ' of ' + self.suit()
13
14    def face_value(self):
15        return Card.FACE_VALUES[self.number % 13]
16
17    def suit(self):
18        return Card.SUITS[self.number // 13]
19
20
21 class CardCollection:
22     def __init__(self):
23         self.cards = []
24
25     def size(self):
26         return len(self.cards)
27
28     def add(self, card):
29         self.cards.append(card)
30
31     def remove(self):
32         return self.cards.pop()
33
34     def __str__(self):
35         return ', '.join( str(card) for card in self.cards )
36
```

```
38 class Deck(CardCollection):
39     # Override ancestor's constructor, i.e. replace the default.
40     def __init__(self):
41         self.cards = []
42         for cardnum in range(52):
43             self.add( Card(cardnum) )
44
45     # Alias the inherited method "size" as "cards_left",
46     # because we usually ask how many cards are left in a
47     # deck rather than asking about its size.
48     def cards_left(self):
49         return self.size()
50
51     # Another alias. When using a deck of cards we talk about "dealing"
52     # cards not "removing" them from the deck.
53     def deal(self):
54         return self.remove()
55
56     # Add a new method, shuffle, that does not exist in ancestor class.
57     def shuffle(self):
58         ncards = len(self.cards)
59         for swaps in range(ncards):
60             swaps = ncards - 1 - swaps
61             posnl = random.randint(0, swaps)
62             self.cards[posnl], self.cards[swaps] = self.cards[swaps], self.cards[posnl]
63
64
65 class Hand(CardCollection):
66     pass
67
```

[demo]

# Polymorphism

- Polymorphism allows same syntax to be used with objects of different types
- Equivalently, the meaning of a given method or operator depends on the type of object applied

```
x = y + 5  
salutation = 'Mr.' + last_name  
t = [3, 7] + [6, 2]
```

- Can we make our custom object types trigger appropriate behaviour?
- Can we give our objects same status and abilities as Python's built-in types?

# Polymorphism

All we need to do is to define some special methods!

When Python encounters an operator in an expression:

- it identifies the objects around it (for binary op.)
- then calls a method from their class definitions (if provided)

Example: When Python encounters '+',

- it checks if object on the left hand side has `__add__()`
- if so, it is invoked

Example:

- `print` looks for `__str__()` method to invoke
- square brackets for list indexing, `x[2]`, look for `__getitem__()`

# Polymorphism

- Python provides about a dozen of these special methods for operations
- The more of them we define, the more our custom type will seem like a built-in type
- a complete list is given at:  
<https://docs.python.org/3/reference/datamodel.html#specialnames>
- today we will focus on a few of these to get an idea about how they work

# Example: Fractions

- Suppose we are working on a suite of programs for carpenters who want to work with fractions, e.g  $\frac{3}{4}$  of an inch,  $\frac{5}{16}$  of an inch and so on.
- Not surprisingly our design of this suite calls for a Fraction class.
- We don't want to just handle the fractions common in carpentry we want a general Fraction class we will be able to reuse in many settings.
- We'd like to be able to do things like:

```
>>> d1 = Fraction(2,5)
>>> print(d1)
2/5
>>> d2 = Fraction(4)
>>> print(d2)
4/1
>>> d3 = d1 + d2
>>> print(d3)
22/5
```

```
>>> if d1 <= d2:
            print('d1 is less than or equal to d2')
else:
            print('d2 is greater than d1')
d2 is greater than d1
>>> print( d1[0], d1[1] )
2 5
>>> print( d1 + 2 )
12/5
>>> print( d1 - 2 )
-8/5
>>>
```

# Example: Fractions

## Reverse engineering specs

```
>>> d1 = Fraction(2,5) → - object constructor takes two arguments:  
>>> print(d1) → numerator denominator  
2/5 → - it defines an __str__ method  
>>> d2 = Fraction(4) → - constructor can hand a single input:  
>>> print(d2) → second input defaults to 1  
4/1 → - it defines an __add__ method  
>>> d3 = d1 + d2 →  
>>> print(d3)  
22/5  
  
>>> if d1 <= d2: → - it defines an __le__ method  
    print('d1 is less th  
else:  
    print('d2 is greater  
d2 is greater than d1  
>>> print( d1[0], d1[1] ) → - it defines __getitem__ method  
2 5  
>>> print( d1 + 2 ) → - __add__ method works when second object is int  
12/5  
>>> print( d1 - 2 ) → - it defines __sub__ method  
-8/5  
>>>
```

# class Fraction

fraction.py

```
1 class Fraction:
2     def __init__(self, n, d = 1):
3         self.num = n # numerator
4         self.den = d # denominator
5
6     def __str__(self):
7         return str(self.num) + '/' + str(self.den)
8
9     def __add__(self, other):
10        if isinstance(other, Fraction):
11            bottom = self.den * other.den
12            top = (self.num * other.den) + (other.num * self.den)
13            return Fraction(top, bottom)
14        elif isinstance(other, int):
15            other = Fraction(other)
16            bottom = self.den * other.den
17            top = (self.num * other.den) + (other.num * self.den)
18            return Fraction(top, bottom)
19
```

`__str__` has to return str type:  
so use `str()` to convert int to str

`__add__` checks the type of thing being added to using `isinstance`  
it converts other from `int` to `Fraction` when necessary

# class Fraction

```
19
20     def __le__(self, other):
21         return self.num*other.den <= other.den*self.num
22
23     def __getitem__(self, key):
24         if key == 0:
25             return self.num
26         elif key == 1:
27             return self.den
28
29     def __sub__(self, other):
30         if isinstance(other, int): → a simpler way to handle int
31             other = Fraction(other)
32         if isinstance(other, Fraction):
33             bottom = self.den * other.den
34             top = (self.num * other.den) - (other.num * self.den)
35             return Fraction(top, bottom)
36
```

a simpler way to handle **int**  
compare with **\_\_add\_\_**

**\_\_getitem\_\_** is the special method called when the interpreter encounters []s,  
i.e. when it encounters the expression obj[key] it calls obj.**\_\_getitem\_\_**(key).

# Programming practice: Vector class

pull git repo, & complete this task via following the example in fraction.py

**v1 = Vector(xlen,ylen)**  
**print(v1)**

yields "xlen i + ylen j" or "(xlen,ylen)", requires `__str__` method

**v1.norm or .norm() should yield**

$\sqrt{xlen^2 + ylen^2}$

**v1 <= v2 True if v1.norm < v2.norm**

requires `__le__` method

**v1 + v2**

requires `__add__` method

**2\*v1**

yields xlen -> 2\*xlen, ylen -> 2\*ylen

requires `__mul__` method

Extra: define `__neg__` which does

$v1 \rightarrow -v1:$   
 $xlen \rightarrow -xlen, ylen \rightarrow -ylen$

Extra: define `__sub__`

# Vector class: solution

vector.py

```
8 from math import sqrt
9
10 class Vector:
11     def __init__(self,xlen,ylen):
12         self.x = xlen
13         self.y = ylen
14         self.norm = sqrt(self.x**2 + self.y**2)
15
16     def __str__(self):
17         return str(self.x) + 'i + ' + str(self.y) + 'j'
18
19     def __add__(self,other):
20         if isinstance(other,Vector):
21             return Vector(self.x + other.x, self.y + other.y)
22
23     def __le__(self,other):
24         return self.norm <= other.norm
25
26     def __mul__(self,other):
27         if isinstance(other,int):
28             return Vector(other*self.x, other*self.y)
29
30 if __name__ == '__main__':
31     v1 = Vector(1,2)
32     print("v1 is: ", v1)
33     print("v1.norm is: ", v1.norm)
34     v2 = Vector(1,3)
35     print("v2 is: ",v2)
36     print("is v1 <= v2?: ", v1 <= v2)
37     print("v1+v2 is: ", v1+v2)
38     print("v1*2 is ", v1*2)
```

# Example: playing cards

- Our first example of polymorphism dealt with **Fractions**, an immutable type like other numbers and strings.
- There are some subtle issues that arise when dealing with mutable objects like container types.
- Our card classes provide a familiar container type for us to work with

# Example: playing cards

We would like to be able to do:

```
cc = CardCollection()  
card = Card(42)  
cc = cc + card
```

As before we define `__add__`

```
class CardCollection:  
    ...  
    def __add__(self, other):  
        self.cards.append(other)  
        return self
```



Why do we need a `return` statement above?  
`self.cards` is mutable, so why bother?

# Example: playing cards

We would like to be able to do:

```
cc = CardCollection()  
card = Card(42)  
cc = cc + card
```



As before we define `__add__`

```
class CardCollection:  
    ...  
    def __add__(self, other):  
        self.cards.append(other)  
        return self
```



Why do we need a `return` statement above?  
`self.cards` is mutable, so why bother?

This is because assignment operator above is expecting something on RHS.  
If `__add__` doesn't have `return` statement, it will return `None` by default.  
Hence, `cc` is assigned `None` value, and is destroyed.

# Example: playing cards

Thanks to inheritance, we can use + on **Hand** and **Deck** objects

```
roxx = Hand()  
roxx = roxx + Card(42)  
roxx = roxx + Card(18)  
print(roxx)
```



```
>>>  
4 of Spades, 6 of Diamonds  
>>>
```

Python's internals are well enough designed that we can chain + operations without having to write any additional code

```
roxx = Hand()  
roxx = roxx + Card(42) + Card(13) + Card(2)  
print(roxx)
```

```
>>>  
4 of Spades, A of Diamonds, 3 of Clubs  
>>>
```

# Example: playing cards

So far we are adding a **Card** to a **CardCollection**,  
but what if we also wanted to be able to add two **CardCollections** together?

```
def __add__(self, other):
    if isinstance(other, Card):
        self.cards.append(other)
        return self
    elif isinstance(other, CardCollection):
        self.cards.extend(other.cards)
        return self
    else:
        print('You can only add cards ...')
```

```
roxx = Hand()
roxx = roxx + Card(42) + Card(13) + Card(2)
print('roxx:', roxx)
chris = Hand()
chris = chris + Card(3) + Card(4)
print('chris:', chris)
new = roxx + chris
print('new:', new)
```

```
>>>
roxx: 4 of Spades, A of Diamonds, 3 of Clubs
chris: 4 of Clubs, 5 of Clubs
new: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
>>>
```

# Example: playing cards

## A subtle point

```
print('Before:')
roxx = Hand()
roxx = roxx + Card(42) + Card(13) + Card(2)
print('roxx:', roxx)
chris = Hand()
chris = chris + Card(3) + Card(4)
print('chris:', chris)
print('After:')
new = roxx + chris
print('new:', new)
print('roxx:', roxx)
print('chris:', chris)
```

```
>>>
Before:
roxx: 4 of Spades, A of Diamonds, 3 of Clubs
chris: 4 of Clubs, 5 of Clubs
After:
new: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
roxx: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
chris: 4 of Clubs, 5 of Clubs
>>>
```

Notice any problems?

# Example: playing cards

## A subtle point

```
print('Before:')
roxx = Hand()
roxx = roxx + Card(42) + Card(13) + Card(2)
print('roxx:', roxx)
chris = Hand()
chris = chris + Card(3) + Card(4)
print('chris:', chris)
print('After:')
new = roxx + chris
print('new:', new)
print('roxx:', roxx)
print('chris:', chris)
```

```
>>>
Before:
* roxx: 4 of Spades, A of Diamonds, 3 of Clubs
  chris: 4 of Clubs, 5 of Clubs
After:
  new: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
* roxx: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
  chris: 4 of Clubs, 5 of Clubs
>>>
```

- **roxx** is changed by what we've done in creating **new**
- The problem: we're building return value from an existing object
- hence
  - existing object is modified
  - new and existing object now referring to the same modified object
- Mutability and shared references can be serious problems

# Example: playing cards

## A subtle point

Solution: create a new **CardCollection** obj in `__add__` method, return this obj

```
def __add__(self, other):
    # Create a new CardCollection.
    new_cc = CardCollection()
    # Put a copy of self's cards into it.
    new_cc.cards = self.cards[:]
    if isinstance(other, Card):
        new_cc.cards.append(other)
        return new_cc
    elif isinstance(other, CardCollection):
        new_cc.cards.extend(other.cards[:])
        return new_cc
    else:
        print('You can only add Cards to CardCollections!')
```

Before:

roxx: 4 of Spades, A of Diamonds, 3 of Clubs

chris: 4 of Clubs, 5 of Clubs

After:

new: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs

roxx: 4 of Spades, A of Diamonds, 3 of Clubs

chris: 4 of Clubs, 5 of Clubs

>>>

# Conclusion: adding new types

**Using classes to add new types to a programming language is a very powerful technique**

**it makes it easier to write programs in domains the language does not support natively.**

**This is a big achievement of object-oriented programming.**

**Adding new types requires paying close attention to the specifics of the situation you are modelling, especially where mutable types are used.**

# **June 20: Wrapping up & Remarks**

## **Final practice**

# Congratulations!

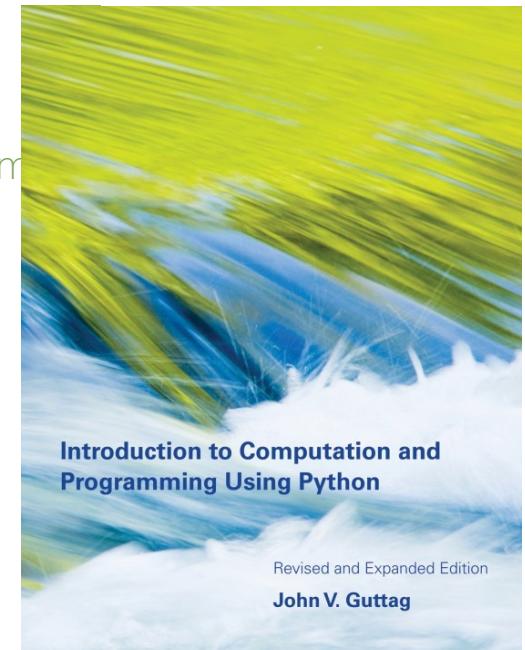
- **You've made it to the end of the course**
- **We've covered a lot of material in the last 2 months**
- **This was a big commitment:**
  - 6 hours of after hours class + assignment every week (on top of full time work for many of you)
  - doing this during spring/summer
- **You should be proud of yourselves!**
- **This will be rewarding:**
  - programming skills are more than ever becoming an integral part of professional work life

# **What we haven't covered (enough)?**

- **recursion**
- **structured testing, testing frameworks  
(unittest)**
- **exception handling (try: ... except:...)**
- **debugging**
- **data structures & algorithms**
- **git**
- **Jupyter**

# Where do we go from here?

- Continuation of this course, CPSC 129, is not available
- Consider taking online courses
- (**edX.org, coursera, Udacity**)
  - introductory Python courses (practice makes perfect!)
  - intro to Computer science courses taught in Python
- Check out other online resources
- Check out textbook by John V. Guttag:
  - <https://mitpress.mit.edu/books/introduction-computation-and-programming-python-revised-and-expanded-edition>



# Where do we go from here?

- **Assign a few hours every week to practice & learn: check out Python programs in the field that you are interested in.**
- **Those who are familiar with Matlab, might want to check out PyLab, Numpy etc**
- **Do you find yourself doing repetitive tasks on computer? Consider these as an opportunity to practice your python skills.**

# Where do we go from here? More on Computer Science

- **Data Structures & Algorithms**
- **Are you curious about hardware aspect of computation? Consider taking a course on:**
  - Computer Architectures or
  - Computation Structures
- **Curious about IT (infrastructure) stuff? Check out Linux Academy: <https://linuxacademy.com/>**
  - Linux, Cloud computing (OpenStack, AWS etc),
  - DevOps, Security, Containers, BigData



Linux Academy

# Tips For Final

- Check out '**Examinable Topics**' section at **github repo**
- **review slides and examples we did**
  - try to program example problems on your own:
- **go over assignments & solutions to assignments**
  - if you have time, try to redo assignments
- **practice, practice, practice!!!**