Review

CS 1MD3 • Introduction to Programming
Winter 2018

Dr. Douglas Stebila

McMaster University

Outline of Topics

- Introduction to Computer Science, Computer Systems, Python programming language, Computational thinking
- Values and types (innate data types, data encoding, expressions, variables, assignment, strings, lists, object & classes)
- Imperative programming (modules, flow of control, control structures -- loops, exceptions and exception processing, procedures and parameter passing)
- Input and output, files, and operations with files
- Data visualization
- Databases and machine learning

ON PROGRAMMING

Declarative vs Imperative Descriptions

Declarative descriptions

- Declarative descriptions state the **properties** of the result. They describe what the result is.
- They are typically short, but cannot be followed.
  - In the case of x-intersect, there could be several possible results or there could be none

Imperative descriptions

- Imperative descriptions tell **how** the result is obtained by given a **sequence of instructions**.
- Each step is simple enough that it can be followed.
  - In the case of x-intersect, restrictions on the input are imposed (f monotonic), additional input is needed (a, b, ε), and the result is only approximate (with ε). There is repetition (line 1) and selection (lines 3, 4).

Algorithms

- An algorithm specifies a **sequence of instructions** to be executed by a **machine** that, when provided with **input**, will eventually stop and produce some **output**.

- Algorithms must be **precise**: each instruction and the next possible instruction to be taken must be unambiguous.

- Algorithms must be **effective**: each instruction must be executable by the underlying machine.

Fundamental questions about algorithms

Correctness

- What output is produced for each input?
- Is it what we "intended"?

Efficiency

- How long does it take to produce the output for a particular input?
  - Measured in terms of time
  - Measured in terms of instructions
  - In the worst case?
  - On average?
- How many resources (like memory) are needed in the process?

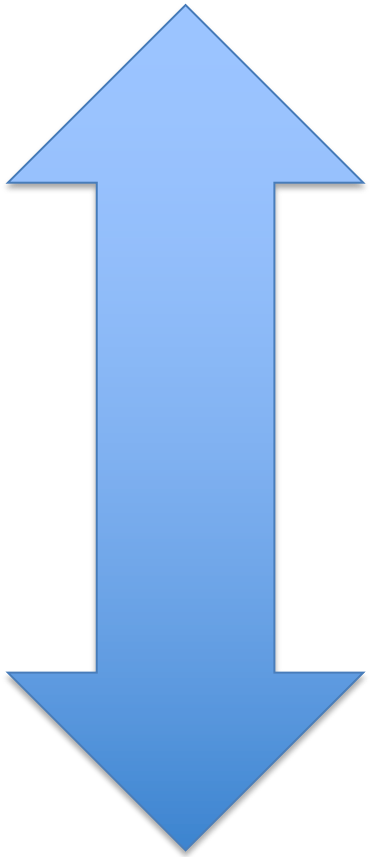
Differences in programming languages

Python
R
Haskell

VisualBasic
Java

C
Assembly

High level



Low level

- Provides programmer lots of abstractions so formulate complex mathematical expressions easily

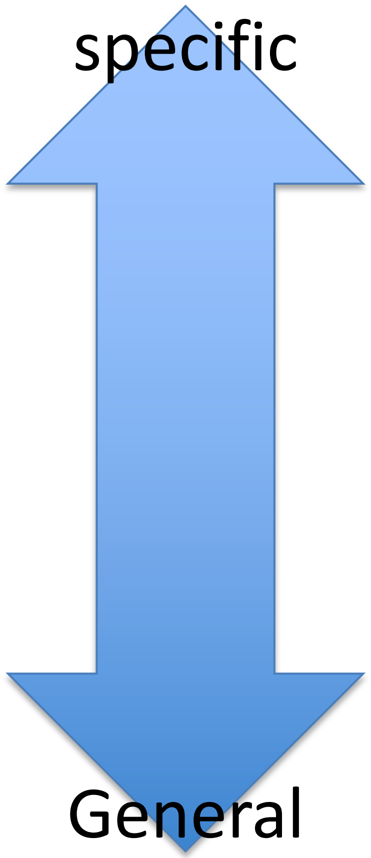
- More direct access to memory and hardware but complex operations require more programmer effort

Differences in programming languages

SQL (databases)
HTML (webpages)
R (statistics)

Application-specific

- Intended for use for a very specific purpose



Python
C
Haskell

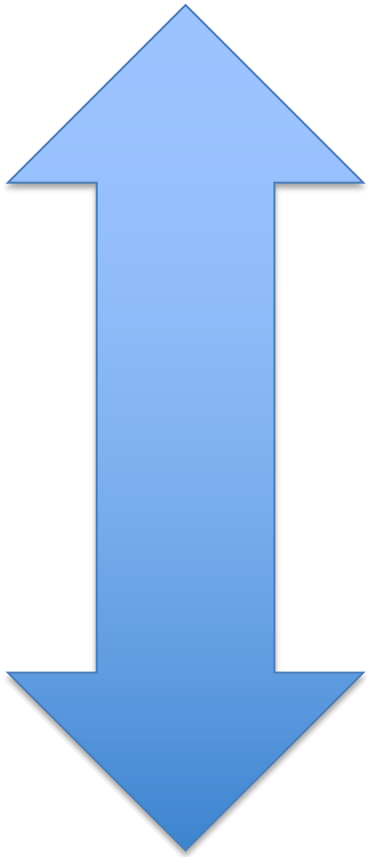
General purpose

- Can be used for many different applications

Differences in programming languages

Python
R

Interpreted



Compiled

Java
C

- Program source code is translated into machine language at runtime (sometimes line by line)
  - Easier to debug

- Program source code is translated into machine language in advance
  - Faster to run

EXPRESSIONS

Basic data types in Python

- `int`: represents positive & negative integers
  - Unlike other languages, no limits to size of an integer
  - Example int literals:
    ```
    3      72    -56789
    ```
- `float`: represents real numbers as floating point numbers
  - Floating point numbers cannot represent all real numbers and do not have perfect precision
  - Example float literals:
    ```
    3.0   72.13   -56789.0123
    ```
- `bool`: represents True and false

Binary operators on `int` and `float`

`a + b`	Addition
`a - b`	Subtraction
`a * b`	Multiplication
`a // b`	Integer division 7 // 4 is 1
`a / b`	Floating-point division 7/4 is 1.75
`a % b`	"a mod b" Remainder from integer division 7 % 4 is 3
`a ** b`	Exponentiation

Boolean operators

and

- "B and C" is True if and only if both B and C evaluate to True

```
>>> B and C
```

or

- "B or C" is True if and only if either B is True, or C is True, or both are True

```
>>> B or C
```

xor

- "Exclusive or"
- "B xor C" is True if and only if one of them is True and the other is False

```
>>> B ^ C
```

Variables

- Variables are names that point to memory in the computer containing objects
- Variable names in Python can contain uppercase and lowercase letter, digits, and _
- **Assignment**, written =, changes the state of a variable
- We can assignment values to multiple values at once

```
x, y = 5, 3
```

Tuples

- Sequence of objects, in order
- Objects don't have to have the same type
- Tuples are immutable

- `>>> (4, 7.0, "potato", True)`

Operations on tuples

T[i]	Retrieves the i+1'th element of the tuple
T + U	Concatenation Yields a new tuple containing all the items in T followed by all the items in U
len(T)	Returns the number of elements in the tuple
x in T	True if x is an element in the tuple T False otherwise
x not in T	True if x is not an element in the tuple T False otherwise

Strings

- Sequence of characters (like a tuple)
- String literals wrapped in single or double quotation marks
- Immutable: can't change individual characters without creating a new string

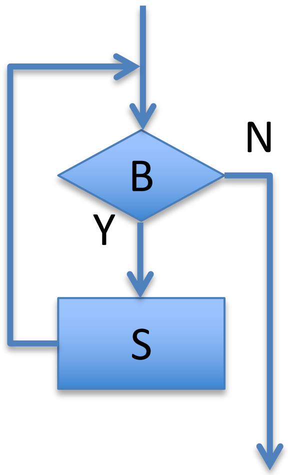
Operators on strings

Operator	Notation
indexing	`S[i]`
suffix, starting at `i`	`S[i:]`
prefix, not including `j`	`S[:j]`
slice, starting at `i`, not including `j`	`S[i:j]`
length	`len(S)`
occurrences of a substring	`S.count(E)`
first index of a subtring	`S.index(E)`

STATEMENTS

Repetition: while loops

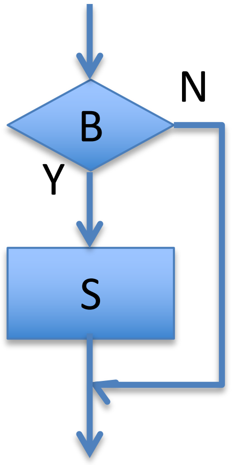
Repetition:



```
while B:
    S
```

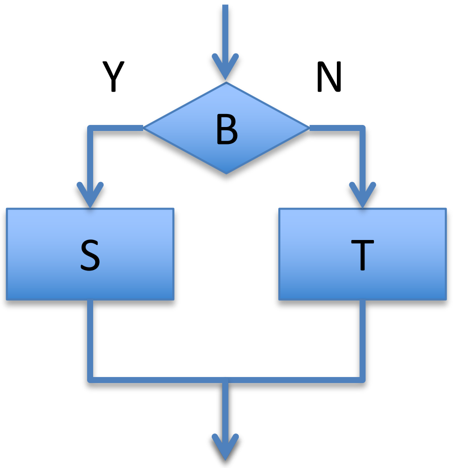
Branching programs: if

Selection:



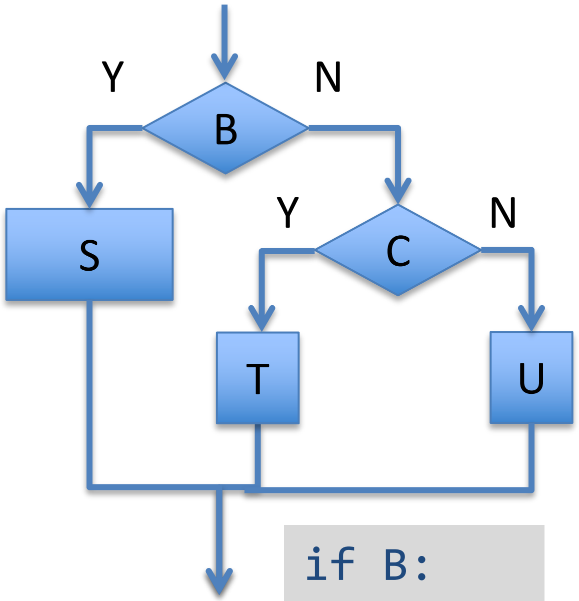
```
if B:
    S
```

Selection:



```
if B:
    S
else:
    T
```

Selection:



```
if B:
    S
elif C:
    T
else:
    U
```

for loops

- for loops are another type of loop structure
- They iterate over all elements of a tuple, string, …
- Unlike while loops, the number of iterations in a for loop is always bound
  - Cannot get into infinite loops*

* Really talented programmers can always get into infinite loops

for loops
Iterating over characters in a string

```
s = "potato"
i = 0
while i < len(s):
    c = s[i]
    print(c)
    i += 1
```

```
s = "potato"
for c in s:
    print(c)
```

The range object

- Generates a sequence of numbers
- Three versions of the range object:
  - range(n)
    - Generates the sequence 0, 1, 2, … n-1
  - range(a, b)
    - Generates the sequence a, a+1, a+2, …, b-1
  - range(a, b, step)
    - Generates the sequence a, a+step, a+2*step, …, z
    - Where z is of the form a+c*step and z < b

Common errors in loops

- **incorrect initialization**, particularly that of accumulators
- **incorrect termination condition** of the loop
  - e.g. < vs. <=
- **infinite loops** by incorrectly updating the loop variable
- index calculation is "**off by one**"
- **improper bracketing/indentation** of nested structures

Nested for loops

- We can nest for loops if we want to iterate combinations of values

- This example combines all adjectives with all nouns:

```
adjectives = ('pretty', 'happy', 'fun')
nouns = ('dog', 'boy', 'girl')
for a in adjectives:
    for n in nouns:
        print("the {:s} {:s}".format(a, n))
```

for vs while Loops

- for loops automatically initialize and increment the loop variable, thus avoid certain class of errors
- for loops may necessitate jumps out of the loop by return or break, thus having multiple exits; while loops allow a single exit and make the termination condition explicit

```
...
while i<b and s[i]!='n':
    ...
    i = i+1
    ...
# i>=b or s[i]=='n'
...
```

```
...
for i in range(len(s)):
    ...
    if s[i]!='n': break
    ...
# ?
...
```

- for this reason, we avoid break and prefer a single return at the end of a function

STRUCTURED DATA TYPES

Lists

Like tuples, **lists are ordered collections**. Elements don't have to be of the same type. Elements can occur multiple times.

Main difference: lists are mutable: they can be updated.

To create a list: use square brackets. [ ]

To create a typle: use parentheses. ( )

```
L = [1, 3, True, "Potato"]
len(L)
print(L[3])
L[2] = False
```

Lists

Operation	Notation
empty list, list with E, list with E, F, …	`[] , [E], [E, F], [E, F, G], …`
list L indexed at I	`L[I]`
suffix of L starting at I	`L[I:]`
prefix of L up to I	`L[:I]`
slice of L starting at I up to J	`L[I:J]`
list L concatenated with M	`L+M`
length of L	`len(L)`
E member of L, not member of L	`E in L, E not in L`
L repeated I times	`L*I, I*L`
occurrences of E in L	`L.count(E)`
first index of E in L	`L.index(E)`

Mutability

Unlike tuples and strings, lists are mutable

```
>>> a = [1, 2, 3]; a[1] = 4; a
>>> a = [1, 2, 3]; a[1:] = [5, 6]; a
```

A number of functions update lists in place

Operation	Notation
append element `E` to list `L`	`L.append(E)`
extend list `L` with list `M`	`L.extend(M)`
insert `E` at index `I`	`L.insert(I, E)`
delete element at index `I`	`L.remove(I)`
return element at index `I` and delete it	`L.pop(I)`
reverse list `L`	`L.reverse()`

Sets in mathematics

- Sets in mathematics are **unordered** collections of objects without repetition
- E.g., the set of all even numbers less than 10:

  {0, 2, 4, 6, 8}

  or equivalently

  {4, 2, 6, 6, 8, 8, 0, 4}

Set operations

Operation	Notation
empty set	`set()`
set with E, set with E, F, …	`{E}, {E, F}, {E, F, G}`
union (∪) of S and T	`S|T`
intersection (∩) of S and T	`S&T`
difference (−) of S and T	`S-T`
cardinality of S	`len(S)`
E element of S, not element of S	`E in S, E not in S`
add E to S	`S.add(E)`
remove E from S	`S.discard(E)`
return and remove arbitrary element	`S.pop()`

Comparing collection types

Tuple	List	Set
• (…)	• […]	• {…}
• Duplicates allowed	• Duplicates allowed	• No duplicates
• Order matters	• Order matters	• Order doesn't matter
• Immutable	• Mutable	• Mutable

Dictionaries

Dictionaries contain **key-value pairs**.

- The index is called the **key**
- The value is called the **value**

```
daysInMonth = {'Jan': 31, 'Feb': 28,
'Mar': 31, 'Apr': 30}

print(daysInMonth['Jan'])
```

Dictionaries

Operation	Notation
empty dictionary	`{}`
dictionary mapping key E to value F, …	`{E: F}, {E: F, G: H}`
looking up key `E` in `D`	`D[E]`
key `E` in `D`, key `E` not in `D`	`E in D, E not in D`
deleting key `E` and its value from `D`	`del D[E]`
size of `D`	`len(D)`
get a tuple of all the keys in `D`	`d.keys()`
get a tuple of all the values in `D`	`d.values()`

FUNCTIONS AND RECURSION

Syntax of functions

def used to define a function

Same rules for naming functions as for variables

0 or more variable names, which will be assigned to whatever values the function caller provides

```
def name_of_function(list_of_inputs):
    body_of_function
    return value_to_return
```

The output to give back to the function caller

Body and return statement have to be indented

Function parameters

```
def change(target, coins):
    # some code here
    return d

t = 185
c = [5, 10, 25, 100, 200]
change(t, c)
```

Formal parameters

Actual parameters

- When a function is defined, the **formal parameters** are included in the function definition
- When the function is invoked (called), the **actual parameters** are provided
  - During invocation, the formal parameters are **bound** to the actual parameters
  - Python uses **pass by assignment** in which each formal parameter is assigned (using =) to the actual parameter

Default parameters

```
def sort(values, ascending = True):
    # your code here
    return whatever

sort(L)
sort(L, True)
sort(L, False)
sort(L, ascending = False)
```

- When defining a function, can provide default values for function parameters
  - Need to have parameters with default values after all parameters without default values
- When calling a function, can omit optional parameters which will then be assigned the default value
  - Careful you don't get confused when there are multiple optional values

Recursion

- Recursion is when a function calls itself

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

factorial(5)
```

Base case

Iterative case (recursion)

- Sometimes recursion allows for a more natural solution to a problem
- But sometimes recursion can lead to a less efficient solution to a problem

Functions as first class objects

- Can pass functions as arguments to other functions

- Can create anonymous functions
  - `lambda x: x**2`

Software design principles

- **Modularity / decomposition**: a program is broken into parts (functions) that are
  - reasonably self-contained
  - achieve a clear purpose, and
  - can be reused
- **Abstraction:** a component (function) of a program can be used without knowing how it achieves its goal

Measuring efficiency

Theoretical

- **Runtime complexity**: what is the approximate number of basic operations the algorithm will perform for a certain set of inputs?
- **Memory complexity**: what is the approximate number of entries that the algorithm will read/write for a certain set of inputs?

Practical

- **Runtime**: how many seconds does this implementation run for on a particular computer with a particular input?
- **Memory**: how many megabytes of RAM does this implementation use on a particular computer with a particular input?

NUMERICAL COMPUTATIONS

Base 10
Decimal representation

2	0	5	3
10^3	10^2	10^1	10^0

$2*10^3 + 0*10^2 + 5*10^1 + 3*10^0$

$= (((((2*10)+0)*10)+5)*10)+3$

$= 2053$

Base 2
Binary representation

1	1	0	1
2^3	2^2	2^1	2^0

$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

= 8 + 4 + 1 (in base 10)

= 13 (in base 10)

Converting from decimal to binary

For converting a number into a series a digits, the remainder of the division with the base gives the least digit; this is repeated with the quotient, until the number fits into a single digit

2053 % 10 = 3

2053 // 10 = 205

205 % 10 = 5

205 // 10 = 20

20 % 10 = 0

20 // 10 = 2

2 % 10 = 2

2 // 10 = 0

$\Rightarrow$ stop

13 % 2 = 1

13 // 2 = 6

6 % 2 = 0

6 // 2 = 3

3 % 2 = 1

3 // 2 = 1

1 % 2 = 1

1 // 2 = 0

$\Rightarrow$ stop

13 decimal = 1101 binary

Other bases

Octal

- Octal = base 8
- Digits from 0 up to 7

Example:

- 135 base 8
  $= 1*8^2 + 3*8^1 + 5*8^0$
  = 93 base 10

Hexadecimal

- Hexadecimal = base 16
- Digits from 0 up to 15
  - Represent "digits" 10, 11, 12, 13, 14, 15 as A, B, C, D, E, F

- Example:
- 2A base 16
  $= 2*16^1 + 10*16^0$
  = 42

Real numbers and floating point

- Not all real numbers have finite decimal or binary representations
- Not all decimal numbers have finite binary representations
- Floating point numbers:
  - Represented as ± (1.f) * 2^e
  - 1 bit for ±, 11 bits for e, 52 bits for f
- Alternatives to floating point:
  - Arbitrary-precision decimal
  - Fractions with separate numerator & denominator

Approximation algorithms: Computing the x-intersect

Input:

- a range [a, b]
- a function f that is monotonically increasing on [a, b] such that $f(a) \le 0$ and $f(b) \ge 0$
- a precision $\varepsilon > 0$:

Instructions:

1. As long as $b - a > \varepsilon$, do steps 2–4
2. Calculate $m = (a + b)/2$
3. If $f(m) \le 0$, set a to m
4. Otherwise, if $f(m) \ge 0$, set b to m

Output:

- Values (a, b) such that $f(a) \le 0$, $f(b) \ge 0$, and $b - a \le \varepsilon$

TESTING AND EXCEPTIONS

Testing

- **Testing**: determine whether a program behaves as intended
- **Debugging**: trying to fix a program that does not behave as intended
- **Black-box testing**: Not relying on seeing the source code, only testing input-output behaviour
- **White-box testing**: Inspecting the source code

Testing

- **Unit testing:** Testing each component (function) individually
- **Integration testing**: Testing the program as a whole

- **Edge cases:** Trying to design tests that cover extreme inputs e.g. 0, empty string, start/end of string

Exceptions

- Exceptions allow Python to deal with situations where it tries to execute a statement that isn't well defined or violates some condition
- Python **raises** an exception
  - This interrupts the normal flow of execution
- There can be an exception **handler** which tries to recover
- Or the exception can be **unhandled** in which case the program crashes

Handling exception

```
def doIt2():
    try:
        a = int(input("Enter a number: "))
    except ValueError:
        print("You did not enter a valid number")

>>> doIt2()
Enter a number: potato
You did not enter a valid number
```

Raising exceptions

- You can **raise** exceptions in your code if it enters a situation your code isn't intended to handle

```
def findFirstEven(L):
  """Returns first even number in list L.
     Raises ValueError if no even number in L."""
  for x in L:
    if x % 2 == 0: return x
  raise ValueError
```

CLASSES

Classes

- Classes allow us to create custom data types
  - Example: custom exceptions for different error situations
  - Example: custom data type for family tree Person with all the right properties – no risk of missing keys in a dictionary

```
class Person:
  def __init__(self):
    self.name = None
    self.children = []
    self.spouse = None
```

Classes

- The abstract data type is called a **class**
- An instance of the class is called an **object**
- Classes can have
  - **members (attributes/properties/instance variables)**
    - In the class: referred to like self.x
    - Outside the class: referred to like p.x
      - No parentheses
  - **methods (instance functions/member functions)**
    - Outside the class: called as functions like p.move(4, 5)

Classes

- Encapsulation
  - Grouping together properties and operations into a single object
- Abstraction
  - Callers don't need to know how a class is implemented in order to make use of it
- Information hiding
  - Callers **can't** see how a class is implemented or access private members of a class
  - **Private** members in Python: prefix with __ (2 underscores)

Example class

```
class Point:
    def __init__(self, x0, y0):
        self.x = x0
        self.y = y0
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
   def scale(self, s):
        self.x *= s
        self.y *= s
   def distance(self, other):
        return ((other.x - self.x)**2 + \
                (other.y - self.y)**2)**.5
```

Overriding operations in classes

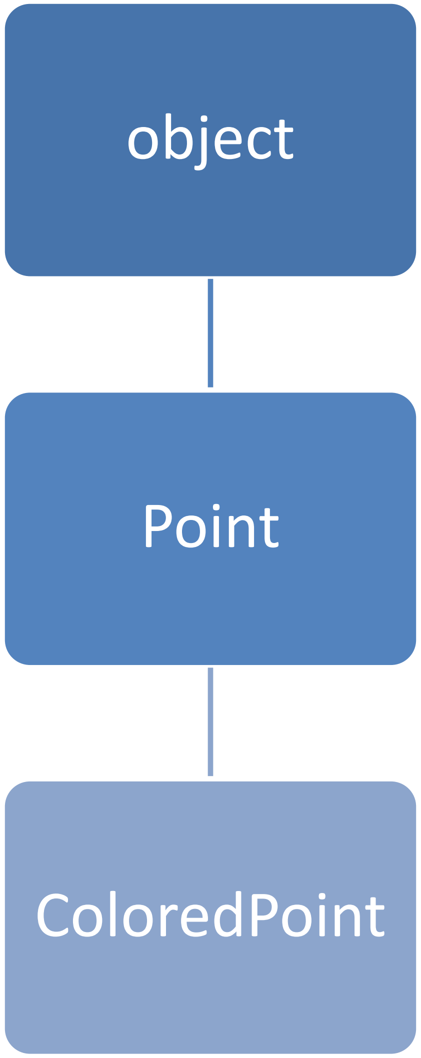
- Can provide methods for certain default operations
  - __str__ for conversion to string
  - __eq__ for equality (a == b)
  - More examples in lab 7

Inheritance

- Can define new classes by **inheritance**: all the definitions of the base class (parent class) are inherited, as if they were copy-and-pasted

- class ColoredPoint(Point) **inherits** all of the methods from Point

- Can **override** individual methods
- Python allows for **multiple inheritance**



FILES

Basic procedure for working with files

1. Open the file for reading or writing to obtain a "file handle".
2. Read or write one or more lines to the file by referring to its file handle.
3. Close the file.

Reading from a file

```
# Approach 1 – Line by line
with open('myfile.txt', 'r') as fh:
   line = fh.readline()
   while line != "":
      line = fh.readline()

# Approach 2 – Line by line
with open('myfile.txt', 'r') as fh:
   for line in fh:
      # do whatever

# Approach 3 – Entire file
with open('myfile.txt', 'r') as fh:
   s = fh.read()
```

Writing to a file

```
with open('myfile.txt', 'w') as fh:
  fh.write("This is a test")
  print("A string", file=fh)
  print("Another string", file=fh)
  print(17, file=fh)
```

Reading data from the Internet

```
import urllib.request
url = 'https://www.douglas.stebila.ca/cs1md3.txt'
response = urllib.request.urlopen(url)

# use the character set specified by the web server
# or try ASCII if none was specified
charset = response.info().get_content_charset() or 'ascii'
s = response.read().decode(charset)

print(s)
```

Not on the exam

Reading CSV in Python

The **csv** module contains functions for working with CSV files

1. Open the file (as in previous lectures)
2. Create a CSV reader object based on the file handle
3. Use a for loop to iterate through the CSV reader object; each line is given as a list
   – If the first line is a header row, we have to put in extra logic to skip it

```
import csv
with open('grades.csv', 'r') as fh:
    grades = csv.reader(fh)
    for row in grades:
        print(row)
```

Reading JSON in Python

The **json** module contains functions for working with JSON data

1. Get the JSON data as a string
   1. Possible by opening a file and then reading from the file handle
2. Use the json.loads to parse the JSON data into a list or dictionary
3. Access the list or dictionary normally in Python

```
import json
with open('potter.json', 'r') as fh:
    s = fh.read()
    x = json.loads(s)
    print("Got a", type(x),
          "of length", len(x))
```

No XML on the exam

Working with dates and times

- The **datetime** module provides classes to represent dates and times, and to manipulate them
  - datetime.date: Class representing dates in Gregorian calendar (y, m, d)
  - datetime.time: Class representing a time on an abstract day of 24*60*60 seconds (h, min, sec, microsecond, timezone)
  - datetime.datetime: Class representing a date and a time on that date
  - datetime.timedelta: Class representing a difference between two time/date objects for arithmetic

Creating a datetime object

```
from datetime import datetime
rightnow = datetime.now()
midterm = datetime(2018, 3, 14, 15, 30)
timeleft = midterm - rightnow
print(timeleft)
print(type(timeleft))
```

No timezones on the exam

Converting datetime objects to strings

- There are many string formatting options to print parts of a datetime object
- Use the **strftime** method to convert from a datetime object to a string
- Use the **strptime** method to convert from a string to a datetime object
  - But need to specify the exact format being parsed

```
print(midterm.strftime("%Y-%m-%d %H:%M:%S"))
2018-03-14 15:30:00
print(datetime.now().strftime("Today is %A"))
Today is Wednesday

christmas = datetime.strptime("Dec 25, 2017 12:00:00", "%b %d, %Y %H:%M:%S")
print(christmas)
print(type(christmas))
2017-12-25 12:00:00
<class 'datetime.datetime'>
```

Don't have to learn all formatting strings

https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior
http://strftime.org/

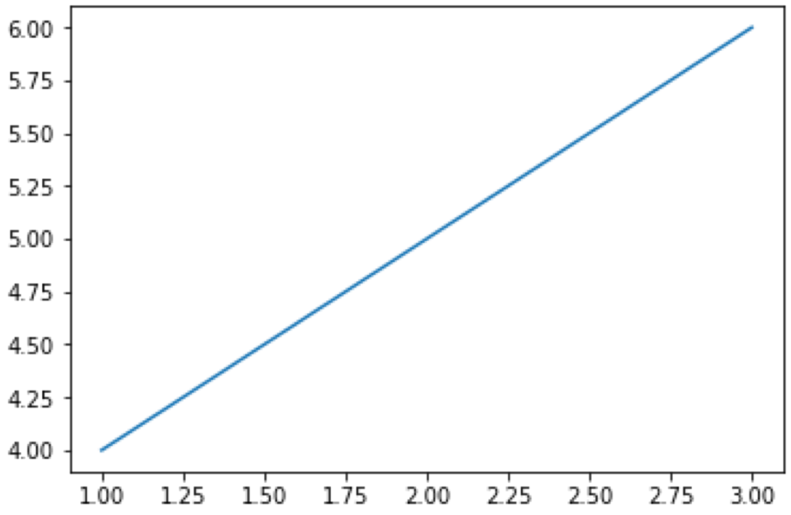
VISUALIZATION

A line plot from a list of points

```
%matplotlib inline

import matplotlib.pyplot as pyplot

xvalues = [1,2,3]
yvalues = [4,5,6]
pyplot.plot(xvalues, yvalues)
pyplot.show()
```



Don't have to customize various aspects of plots on exam

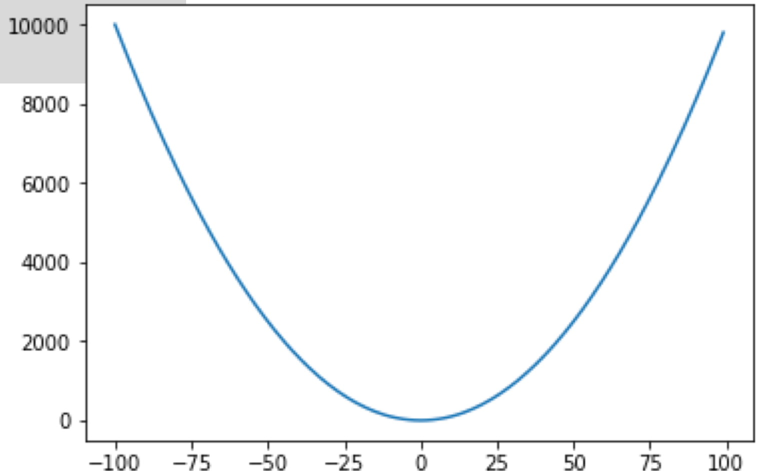
No plot types other than line and scatter plot on exam

A line plot from a function

- To create a line plot where the y values are computed using a function, we have to create a list of the y values by evaluating the function on the x values

```
xvalues = range(-100, 100)
yvalues = [x**2 for x in xvalues]
pyplot.plot(xvalues, yvalues)
pyplot.show()
```

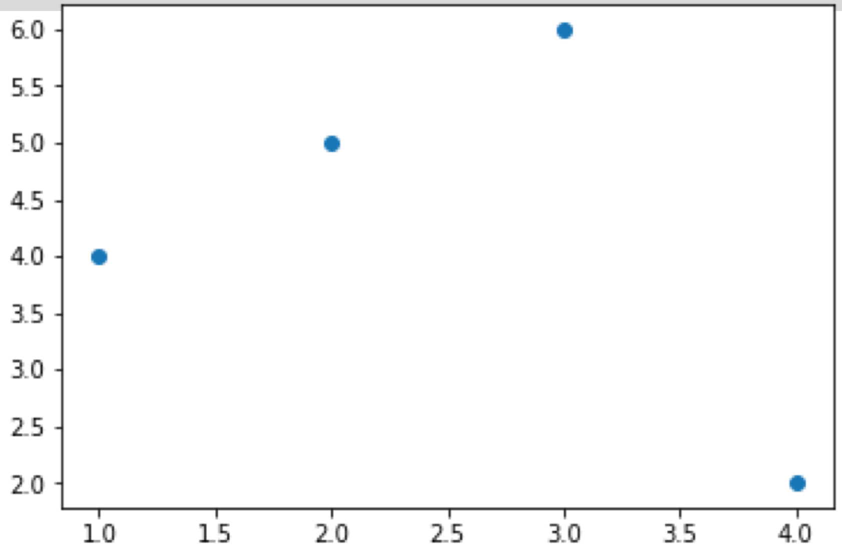
Use our custom frange function to create a range stepped by floating points



Plotting data from a CSV file

```
import csv
with open('sample.csv', 'r') as fh:
    rows = csv.reader(fh)
    xvalues = []
    yvalues = []
    for row in rows:
        xvalues.append(row[0])
        yvalues.append(row[1])
    pyplot.scatter(xvalues, yvalues)
    pyplot.show()
```

```
1,4
2,5
3,6
4,2
```



DATABASES

Databases

- **Relational databases:** Data items and their relationship is stored in tables.
- A **table** is a collection of **records** (a.k.a. rows, objects, entities) with **fields** (a.k.a. values, columns, attributes).
- A **database schema** specifies the names and types of the fields of each table
  - INTEGER: up to 8 bytes integers
  - REAL: 8 byte floats
  - TEXT: Unicode strings
  - BOOLEAN: stored as 0 and 1
  - DATE: stored as numeric value

Structured Query Language (SQL)

- `CREATE TABLE movies (movieid INTEGER, title TEXT, genre TEXT, rating TEXT);`
- `INSERT INTO movies (movieid, title, genre, rating) VALUES (101, 'Casablanca', 'drama romance', 'PG');`
- `SELECT title, movies FROM movies WHERE genre = "comedy";`

No queries other than INSERT and SELECT on exam

Joining related tables using SELECT

rentals

customerid	movieid	daterented	datedue
103	104	2018-2-12	2018-3-12
103	5022	2018-2-28	2018-3-28

customers

customerid	name	address
101	Dennis Cook	123 Broadwalk
102	Doug Nickle	456 Park Place
103	Randy Wolf	789 Pacific Ave.

Returns all fields from both tables
but only in rows which match on customerid

```
SELECT * FROM rentals, customers WHERE
customers.customerid = rentals.customerid
```

customerid in the customers table

customerid in the rentals table

SQLite3 in Python

1. Open the database to get a database handle
2. Do operations on the database using the database handle
3. Close the database handle

```
import sqlite3
db = sqlite3.connect('videostore.db')
cur = db.cursor()
cur.execute('SELECT * FROM movies')
print(cur.fetchone())
print(cur.fetchall())
db.close()
```

MACHINE LEARNING

Types of machine learning

Supervised learning

- Computer is given example inputs and the desired output ("**label**") for each input
  - Examples are called "**training data**"
- Goal: come up with a rule that maps inputs to outputs that performs well even for inputs outside the training data

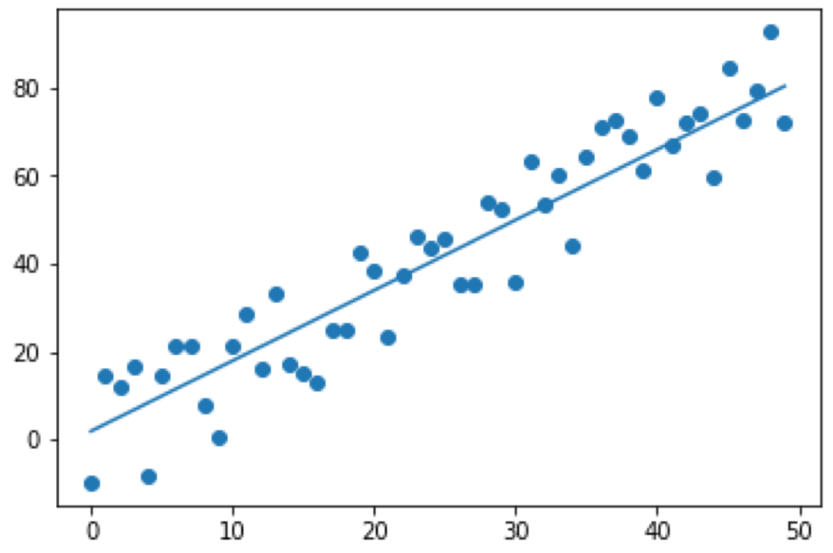
Unsupervised learning

- Computer is given inputs without any labels
- Goal: identify some structure / patterns in the data

No machine learning programming exercises on exam

Regression

- A type of supervised learning
- In regression, output values are continuous (e.g., real numbers) rather than discrete (classes)
- Goal: come up with a rule that predicts the output values based on the values in the training data



Linear regression procedure

1. Obtain data
2. Divide the data into two sets: a training set and a testing set
3. Run linear regression on the training data to get proposed coefficient and intercept
4. Use the proposed coefficient and intercept to predict the y value of the testing data from its x values
5. Measure how far the predicted y values are from the real y values in the testing data set

Classification

- Usually a type of supervised learning
- There are two or more classes / labels
- Goal: come up with a rule that classifies inputs based on the classes in the training

Classification procedure

1. Obtain labelled data
2. Construct feature vectors for the labelled data
3. Divide the data into two sets: a training set and a testing set
4. Train a classifier on the training data to develop a model
5. Use the classifier to predict the labels for the testing data
6. Measure the accuracy of the predicted label on the testing data