#6.100A in 2 Hours

Michael Samuel and Ignacio Guzman

#Class Structure

Lecture will be divided into 4 sections:

- Data, Structures, and Iteration
- Algorithms and Time Complexity
- Functions and Recursion
- Classes and Object Oriented Programming (OOP)

Key Info:

- We will be alternating every other section, with the other being at the “help desk”. Feel free during lecture to stand up and approach the “help desk” for any questions you have from the previous or current sections.
- Also feel free to submit questions to the slido at any time
- You will be able to find the Slides and files after this class on the Splash website.

#Section 1: Data Structures and Iteration

Data Structures

Brief Introduction to Objects

Objects have a type that defines their characteristics and the functions that can operate with them.

Scalar vs. Non-Scalar Objects:

- Scalar objects are the “simplest objects” while non-scalar objects have internal structures
- Scalar Objects: int, float, bool, NoneType
- Non-Scalar Objects: Strings, Lists, Tuples, and Dictionaries
    - Compound Data Types: Strings, Lists, and Tuples
        - Indexed sequence of elements whose elements may be compound data types themselves.

Operators and Expressions

Operators on Scalar Objects:

- Addition (+), Subtraction (-), Multiplication (*), Float Division (/), Floor Division (//), Modulus (%), Exponentiation (**)

To combine objects and operators, you can use expressions. Expressions have a value and a type depending on the objects.

The form is: <object> <operator> <object>

Variables, Assignment, and Binding

Programming Variables != Math Variables

- The former stores a single value while the latter is abstract and can represent many values.

To assign a value to a variable, you use the equal sign, “=”. This binds the variable object to the value object in computer memory. (Distinct objects)

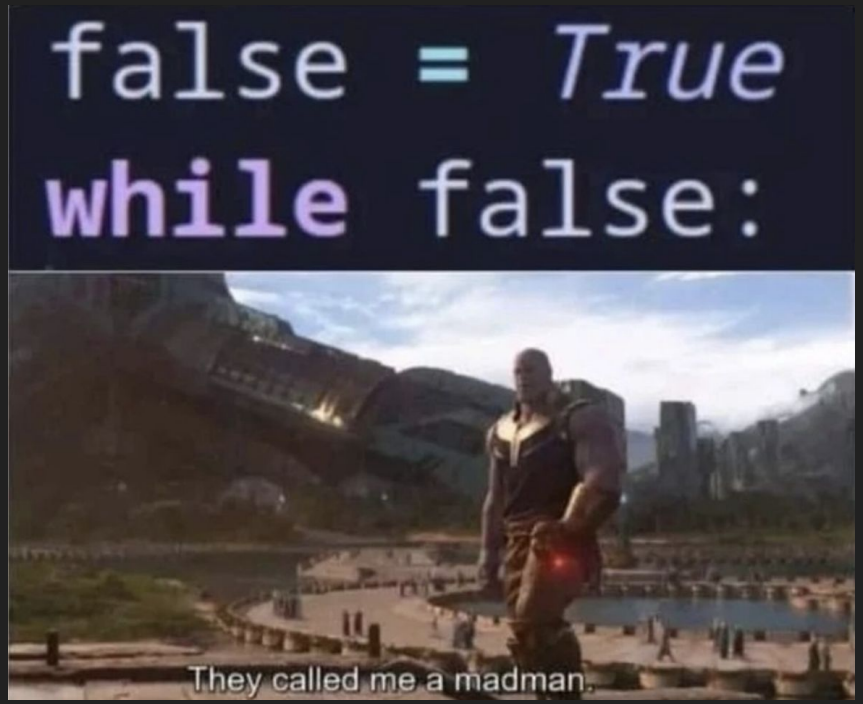
- Variables can be re-binded through new value assignment statements.

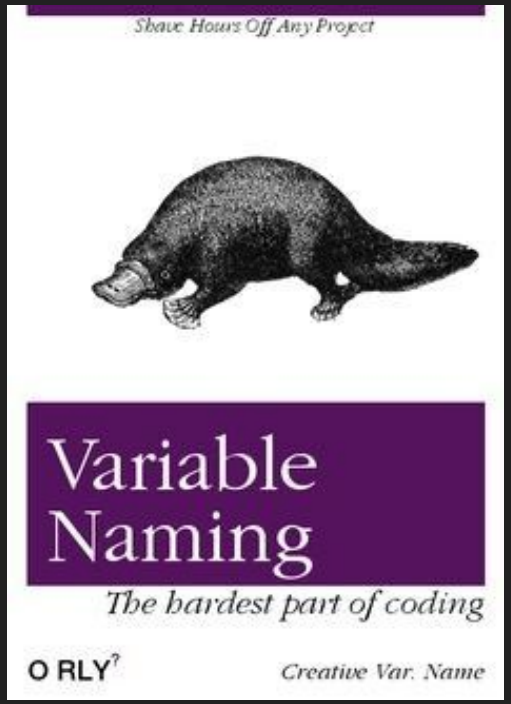
Floats and Approximations

Binary is used by computers since it allows computer hardware to store information as 0s and 1s.

- For integer numbers, binary is easy to use; however, non-integers make it difficult to use.
  - To combat this they used floating point numbers that carry significant digits and an exponent
    - Ex: 6.25 = 25 * 2^-2 = 25/4
    - If the number cannot be rewritten in the form x*(2^p) then it will always be represented as an approximation
- To test floats, never use == as a result of approximations. To test use an epsilon.

Variable Naming





Inputting and Printing

Input:

To assign input from the user and bind it to a variable, you would use the format:

<object> = input(request_message) # request message is a string

Warning:

>>> type(<object>)

str

Print:

To explicitly display results, you would use the format:

print(<object>, …)

Strings (str)

Strings are a sequence of case sensitive characters that are enclosed in either “” or ‘’.

Operator Overloading - When an operator applies to objects that are of different types

- Ex: Concatenation

String Operations:

- len() => gives the length of a string
- Comparison operators: <, >, ==, and others
  - Uppercase < lowercase since based on Unicode order
- Binary Operator: in

Indexing, Slicing, and F-Strings

Strings can be indexed, meaning that characters or strings can be retrieved by referring to their index.

Slicing

- Format: [start:stop:step]
  - [start:stop] if step = 1
  - Characters are retrieved up to “stop-1”th index
  - Helpful ones to remember:
    - s[:] == s[0:len(s):1]
    - s[-1:-(len(s)+1):-1] #reflects the string

We can add variables into strings using concatenation; however, this does not take into account on how it is displayed.

- F-Strings take care of this by allowing you to insert {object} into your string.
  - {object:,.yf} # y is a natural number (i.e. 0, 1, 2,...)

Tuples

Tuples are immutable indexable ordered sequences of objects. Just like with strings, they are indexed and sliced in the same manner.

Notes:

- te = ()
- ts = (2,) # (2,) != (2)

Lists

Lists are mutable indexable ordered sequences of objects (a.k.a. Mutable tuples). Just like with strings and tuples, they are indexed and sliced in the same manner.

If L = [,,,]

- len(L) # gives length
- max(L) # gives maximum value
- L.append(<object>) # adds the object to the list, but returns None
- L.sort() # mutates L and returns None
    - sorted(L) # returns a new sorted list without mutating L
- L.reverse() # mutates L, but returns None
- L.extend(<list>) # adds a list and mutates L
- del L[index] # deletes the element at that index and mutates L
- L.remove(element) # removes the first occurrence of the element, mutates L
- L.pop() # removes the last element of L and returns that element; index can be specified, mutates L

Lists ⇔ Strings

Strings can become lists by calling: list(s) making every character an element of the list.

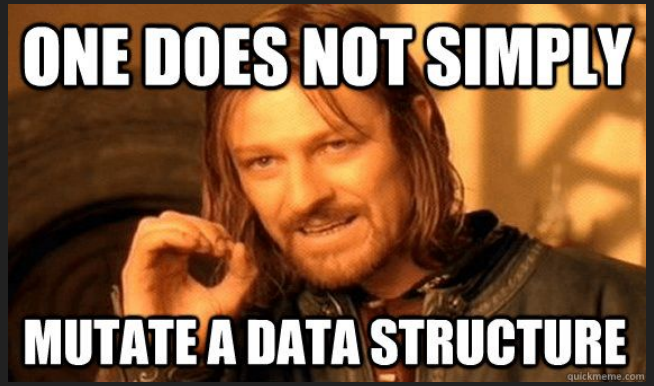
Strings can be splitted on a character through s.split(<string>)

‘’.join(L) joins elements of the list, L, based on the character between the single quotes.

Mutable and Immutable

Strings and tuples are immutable meaning that they cannot be modified. Attempting to change it will result in an error. To avoid this, you can bind your variable to a new object.

This was implemented by the developers because it is good to know that you can have information without the fear of it being changed at some point in your program.



Copying Levels 0 and 1

Level 0: Aliasing

- Aliasing involves having two objects pointing to the same object in memory. Changes to the object in memory change the objects that are pointing to it.

Level 1: Shallow Copy

- This is where we create a copy of a list and not just having a shared pointer. This means that a new data structure is created, but elements are shared.
    - import copy
    - L1 = [...]
    - L2 = copy.copy[L1]

Copying Level N

Level N: Deep Copy

- Deep copying improves upon Level 1 by creating clones at every level of the original structure. Making any changes to the first structure would not affect the second structure.
  - L1 = [...]
  - L2 = copy.deepcopy(L1)

Dictionaries

At the moment, we have learned lists and tuples which allow us to store information. However, When we want to store information that is tied together, we cannot group this without creating more lists/tuples.

To combat this issue, we have dictionaries.

Dictionaries store pairs of data: key and value.

- dictionary1 = {<objecti>: <objecti>}

Dictionaries Lookup and Operations

To return the value associated with a key, we look up the key by doing the following:

- dictionary1 = {...}
- dictionary1[<keyi>] # returns the value or KeyError

Operations

- Add an entry: dictionary1[<new_key>] = <new_value>
- Test if a key is in dictionary1: Use “in”
- Delete an entry: del dictionary1[<key>]
- Change an entry: dictionary1[<desired_key>] = <new_value>

Dictionary Operations Continued

To obtain the keys of the dictionary, we do:

- dictionary1.keys() # returns dict_keys([...])

To obtain the values of the dictionary, we do:

- dictionary1.values() # returns dict_values([...])

To obtain the items of the dictionary, we do:

- dictionary1.items() # returns dict_items([...])

Branching: Booleans and Conditionals

The boolean type has two values: True and False

- Logical Operators: not, and, or
- Comparison Operators: >, <, >=, <=, ==, !=

Conditionals:

- if <condition>:
  - <expression> # or “pass”
- elif <condition>:
  - <expression>
- else:
  - <expression>

Iteration


200,000 units are ready, with a
million more well on the way

Control Flow

While Loops:

- while <condition>:
  - <expression>
  - ...

For Loops:

- for <variable> in <sequence of values>:
  - <code>
  - range(start, stop, step)
    - You can omit start and step if 0 and 1, respectively
- break statement

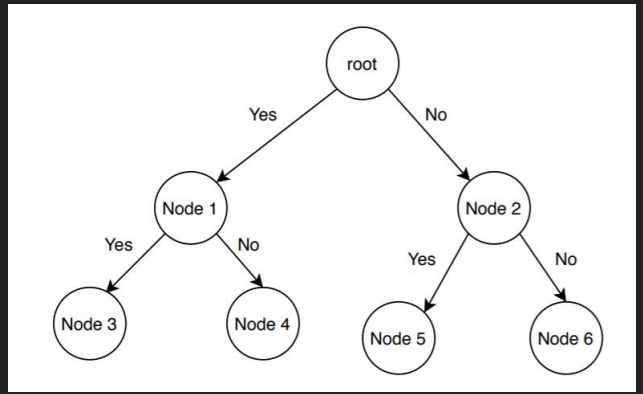
For Loops	Both	While Loops
- Fixed number of iterations - Can easily be rewritten as a while loop	- Both can end early with a break statement	- Unbounded number of iterations - Needs an initialized counter - Cannot be easily rewritten as a for loop

#Section 2: Algorithms and Time Complexity

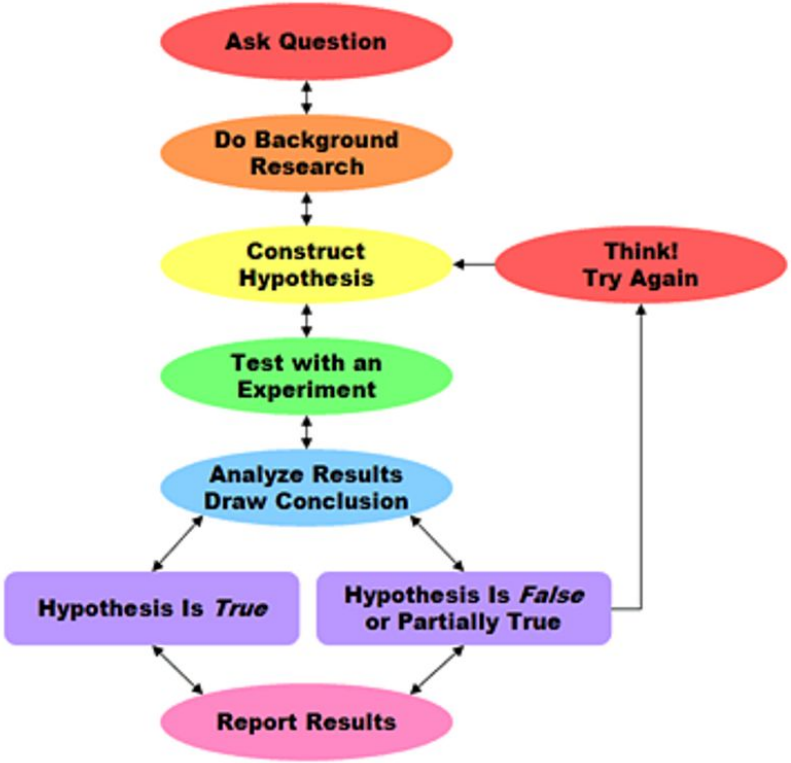
Algorithms

In short, an algorithm is an ordered finite sequence of steps created to solve one or more problems, extract certain qualities from input, or produce a certain result.

Branching refers to the structure of most algorithms– specifically, the “paths” an algorithm may take at each of its “junctures,” or conditionals, are considered “branches” of the algorithm.



Scientific Method



Search Algorithms

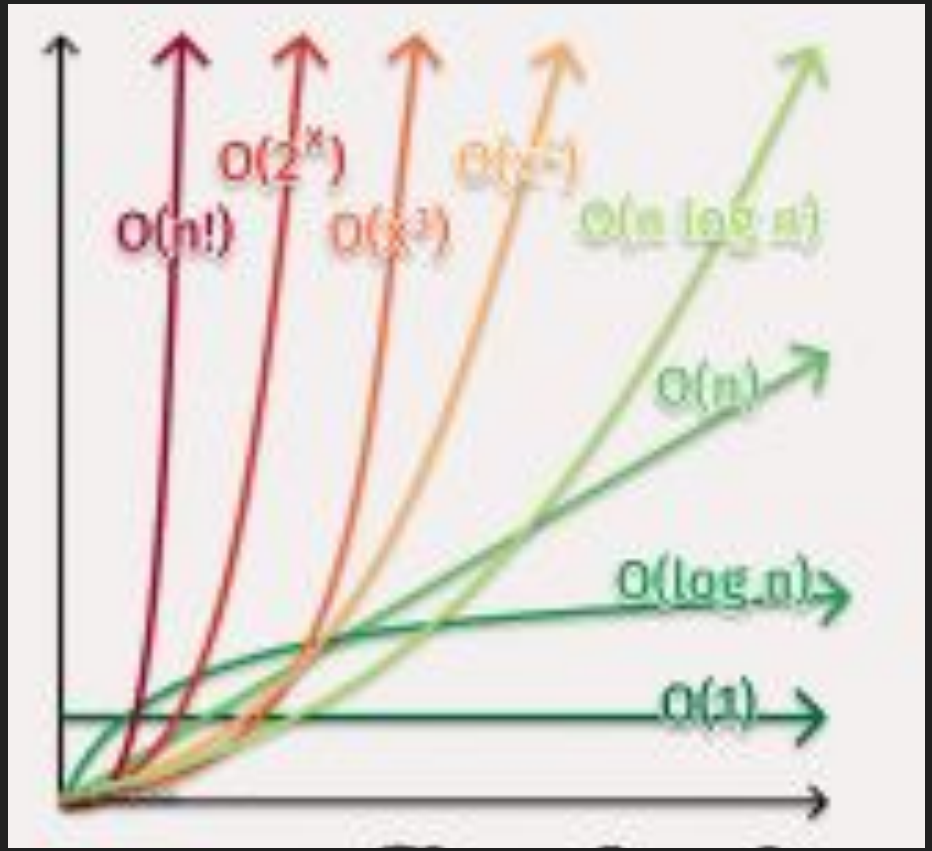
```
68  # simple search
69  ordered_list = list(range(1, 100))
70
71  number = 61
72  i = 0
73  while ordered_list[i] != number:
74      i += 1
```

```
51  # bisection search
52  ordered_list = list(range(1, 100))
53
54  number = 61
55  i= len(ordered_list)//2
56  lower = 0
57  upper = 100
58  search = 0
59  while ordered_list[i] != number:
60      if ordered_list[i] > number:
61          upper = i
62          i = (i + lower)//2
63      else:
64          lower = i
65          i = (i + upper)//2 + 1
```

Time Complexity

We use “big-O notation” to represent the order/magnitude of the worst case number of steps, and thus runtime, of an algorithm.

- We don’t care about coefficients or constants
- Only take the term with the biggest limit



Time Complexity

Common:

O(1) – index calling

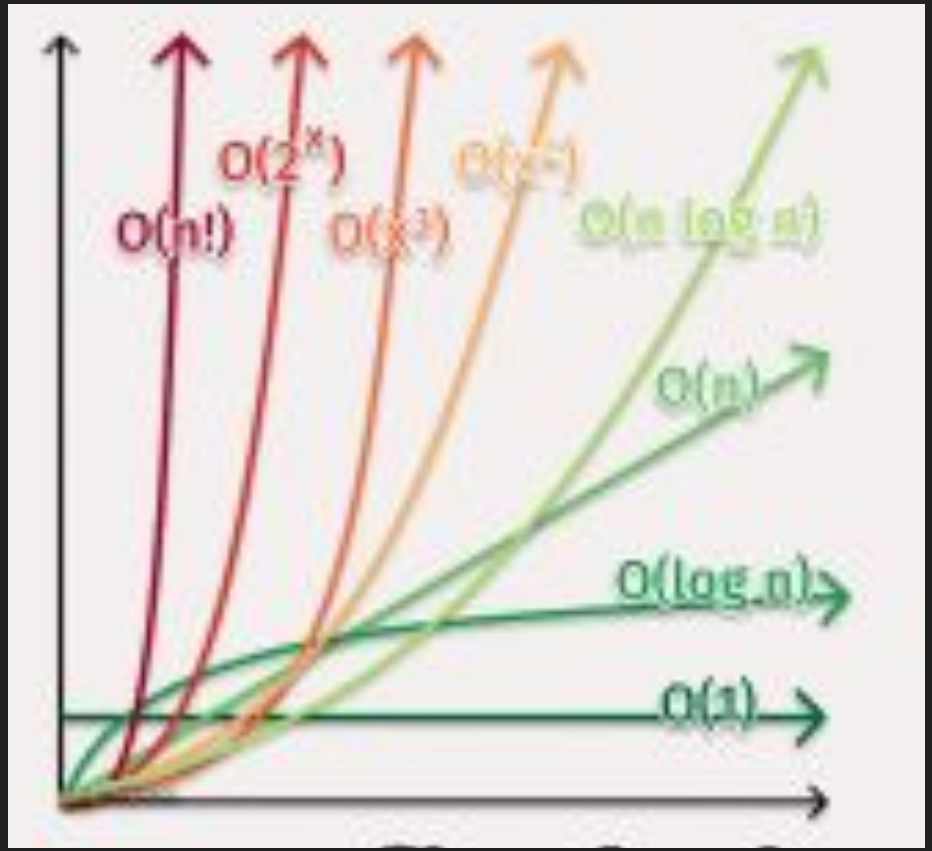
O(log n) – search (e.g. binary)

O(n) – summing a list

O(n log n) – various sorts

$O(n^2)$ – iterating over a list twice (comparison of elements)

$O(2^n)$ – binary tree traversal (height n)



Complexity of Search Algorithms

```
68  # simple search
69  ordered_list = list(range(1, 100))
70
71  number = 61
72  i = 0
73  while ordered_list[i] != number:
74      i += 1
```

```
51  # bisection search
52  ordered_list = list(range(1, 100))
53
54  number = 61
55  i= len(ordered_list)//2
56  lower = 0
57  upper = 100
58  search = 0
59  while ordered_list[i] != number:
60      if ordered_list[i] > number:
61          upper = i
62          i = (i + lower)//2
63      else:
64          lower = i
65          i = (i + upper)//2 + 1
```

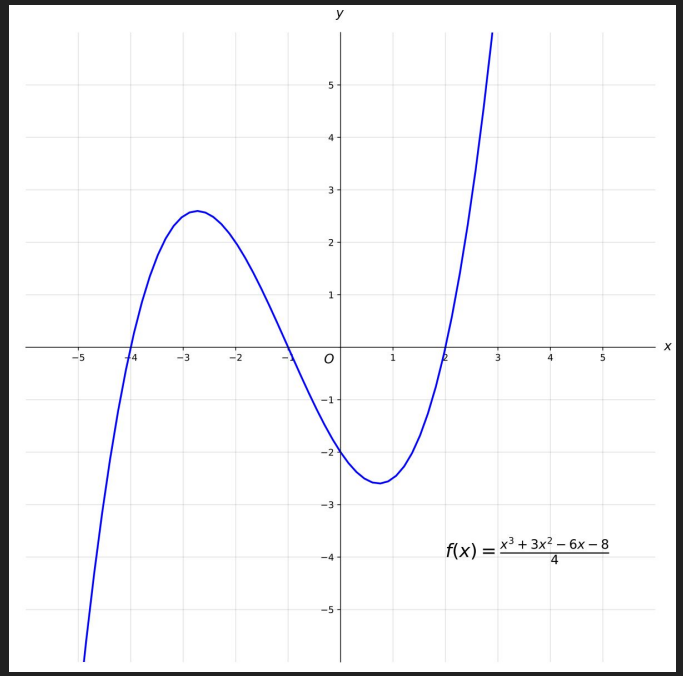
Hidden Complexity

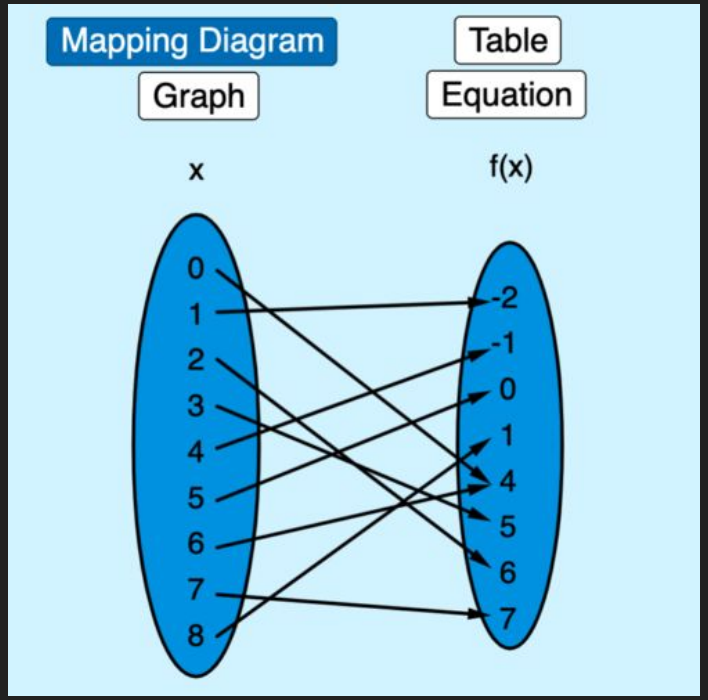
```
43    # sum function
44    list_sum = sum(list)
45
46    # loop summing list
47    for element in list:
48        list_sum += element
```

Functions

What is a function? (math)

A function is some series of mathematical processes that takes **input** (one or several) and produces one **output**.



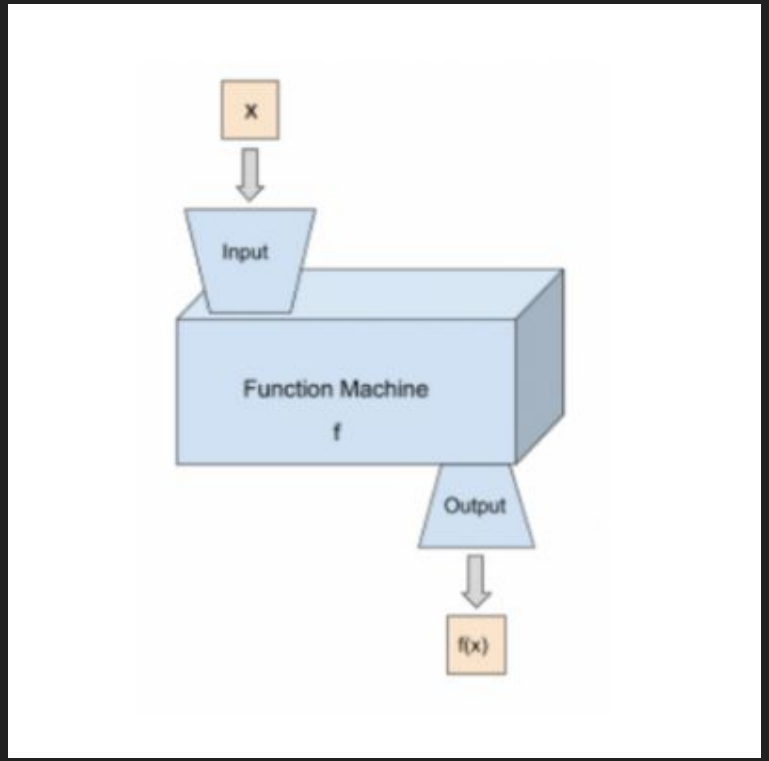


What is a function?

A function is some series of steps or transformations that may take **input** and must produce **output**.

In python, inputs to a function are optional

- Note that they may still use random processes, so they are non-deterministic.
- Functions must produce output. If you do not provide a return value, the function will return None.



Parameters and return values

In python, functions are declared with a keyword called “def” which is short for define.

On the right is the syntax.

In python, functions are procedures, and can sometimes be treated like objects, as both inputs and outputs.

```
77    # f(x) returns 2x - 7
78
79    def func(parameter):
80        # any fancy code here
81        r = 2*parameter
82        return r - 7
```

Function Example

```
 9    def search_for(number, list):
10        while list[i] != number:
11            i += 1
12        return i
13
14    L = list(range(100))
15    search_for(61, L)
```

Scope

Functions cannot access all variables in a code. They can only access variables they have been given, variables they have declared, or variables otherwise within their scope.

- Also true in classes, which you will see later.

```
2    def func():
3        y = x + 5
4        return y
```

Lambda Functions

- Also called anonymous functions
- Functions expressed in “one line” without definition


```
17  # syntax for storing lambda functions
18  Function = lambda x: f(x)
19  Function = lambda x: x**2 #example
20  #when you want to use it later:
21  print(Function(5)) # prints 25
22
23  # you can also create it and use it in the same line (single-use)
24  print((lambda x: f(x))(value))
25  print((lambda x: x**2)(5)) # example (prints 25)
26
27  # more general definition
28  Function = lambda a, b,... z: f(a, b,... z)
```

Example Code (Functions)

```
1
2   # game where you guess what function is being applied to x and y
3   # x is given
4
5   def function_guessr(f):
6       x = 4
7       print("Let's play functionGuessr")
8       while x > 0:
9           print(f"x is {x}.")
10          y = int(input(f"What input do you want to try in the function with x? "))
11          print(f"The function applied to {x} and {y} evaluates to {f(x, y)}.")
12          x -= 1
13
14  func = lambda a, b: a*b
15  function_guessr(func)
```

Recursion

Recursive vs Explicit Definitions

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2)
\end{aligned}
$$

$$
F(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]
$$

Recursive vs Explicit Definitions

$$y_n = y_{n-1} + 1$$

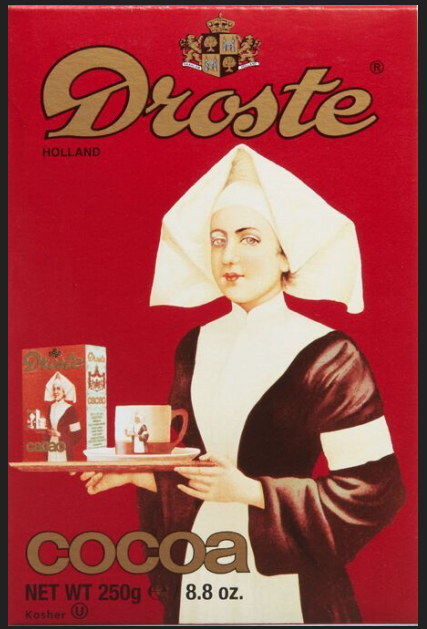
$$y = x$$

Recursion vs Iteration in Programming

Iteration is just like before – for and while loops
- In comparison to recursion, focus on while loops

Recursion – at its base, the process of letting a function call itself

These are interchangeable processes



Something's not right here…

```
3
4   #should print x^2, then print x* user input for x from 10 to 1
5   x = 10
6   while x > 0:
7       print(f"The square of {x} is {x**2}.")
8       y = int(input(f"What should we multiply {x} by: "))
9       print(f"{x} times {y} is {x*y}.")
10      y -= 1
```


Learn to program
Make recursive function
No exit condition
Learn to program
Make recursive function
No exit condition
Learn to program
Make recursive function
No exit condition

The Right Way

- Steps:
- Base case / exit condition – case of function input with non-recursive return statement
- Recursion case – often an “else” for things that don’t meet the exit condition, calls the function again with
  - Transformation of input – extremely important
  - Retention of previous data

Example Code (Recursion)

```
34    # n factorial
35
36    def factorial(n):
37        if n == 0:
38            return 1
39        else:
40            return n*factorial(n-1)
```

#Section 4: Classes and object Oriented Programming (OOP)

What are objects? How is OOP useful?

Objects have a type that defines the characteristics of the programs that work with them. They are are instance of a type.

A type captures an internal representation and an interface. This means that it has data attributes and defined behaviors.

OOP groups data together into packages and allows for divide-and-conquer development. This allows for code to be reused through classes.

Basics of Classes, Attributes, and Methods

Creating a class is very different from using an instance of the class.

- Creating entails defining its name and attributes
- Using an instance entails creating and doing operations on new instances.

Attributes are the “characteristics” that belong to a class. (Ex: age and height of a person)

Methods are the functions that work within the class and handle the attributes.

```
class Coordinate:
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return f"<{self.x},{self.y}>"
```

```
class Circle:
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
    def __str__(self):
        return f"circle: {self.center}, {self.radius}"
    def is_inside(self, point):
        return point.distance(self.center)<self.radius
```

Credit to 6.100A Lecture Notes

Hierarchies and Inheritance

Credit to 6.100A Lecture Notes

Parent Class > Child Class

- Child class inherits the parent class's data, behavior, and methods.

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return f"animal:{self.get_name()}:{self.get_age()}"
```

Equivalent in Python 3:
class Animal:
class Animal():
class Animal(object):

- everything is an object
- class object implements basic operations in Python, like binding variables, etc

```
class Person(Animal):
    def __init__(self, name, age):
        super().__init__(age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.get_age() - other.get_age()
        print(abs(diff), "year difference")
    def __str__(self):
        return f"person:{self.get_name()}:{self.get_age()}"
```

parent class is Animal

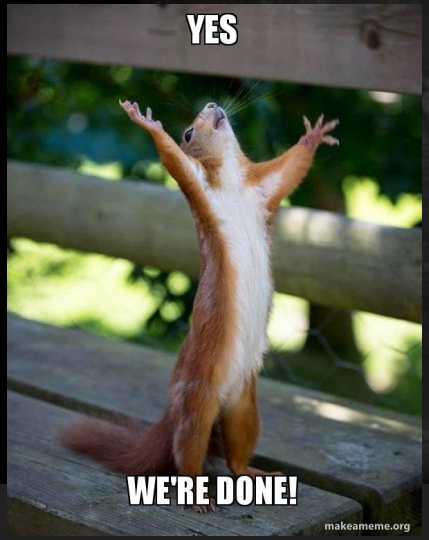
call Animal constructor
same as: Animal.__init__(self, age)
super() is preferred since its generic
call Animal's method

add a new data attribute

new methods

override Animal's __str__ method

The End



Final Remarks

Thank you for attending our class! We hope that you learned something through this rushed version of 6.100A.

Best of wishes to the current seniors and their applications for college. You got this!

Exit Survey:

- https://forms.gle/MAZRk6LnY25rbGrD9

Last but not least… raffle time!