Introduction to Computer Science and Programming Using Python

MITx - 6.00.1x

Summary

Week 1: Python Basics

A: Introduction to Python

Knowledge

Video summary:

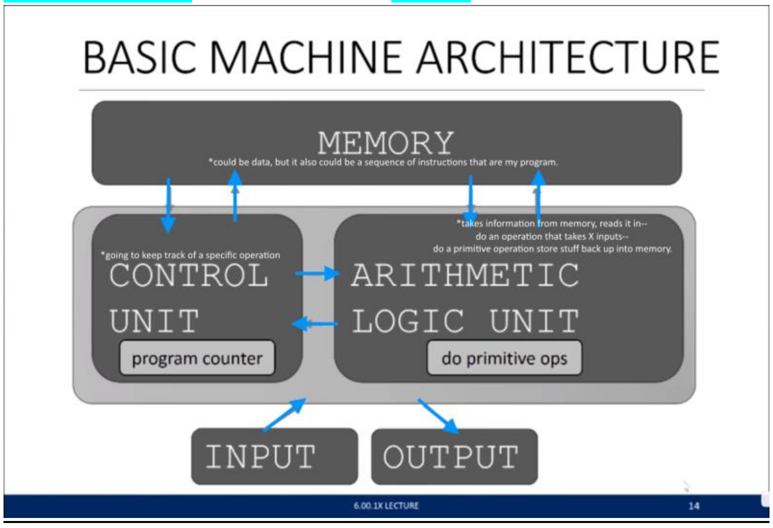
- Declarative knowledge is a statements of fact || imperative knowledge is a recipe or how-to.
- Example of imperative knowledge, Alexandria of Heron's algorithm for square root.
- Imperative knowledge has sequence of simple steps; flow of control; a stop.

Quizz summary:

- An algorithm is a conceptual idea, a program is a concrete instantiation of an algorithm.
- A computational mode of thinking means that everything can be viewed as a math problem involving numbers and formulas.
- Computer Science is the study of how to build efficient machines that run programs
- The two things every computer can do are perform calculations and Remember the results
- Declarative knowledge refers to statements of fact.
- Imperative knowledge refers to 'how to' methods.
- A recipe for deducing the square root involves guessing a starting value for y. Without another recipe to be told how to pick a starting number, the computer cannot generate one on its own.

Machine

Video summary:

- Fixed program = computer like calculator or Alan Turin's Bombe (a computer designed specifically to calculate a particular computation.)
- Stored Program = computer like modern one (machine stores and executes instructions)
- 
BASIC MACHINE ARCHITECTURE
MEMORY
*could be data, but it also could be a sequence of instructions that are my program.
*going to keep track of a specific operation
CONTROL UNIT
program counter
*takes information from memory, reads it in--
do an operation that takes X inputs--
do a primitive operation store stuff back up into memory.
ARITHMETIC LOGIC UNIT
do primitive ops
INPUT
OUTPUT
6.00.1X LECTURE
14
- Sequence of instructions stored inside computer can be built from predefined set of primitive instructions || can be an interpreter who executes each instruction in order (use test to change flow of control; stop it when done)
- Turing showed you can compute anything using the 6 primitives of Turing Complete.
- Indeed modern programming language have more built in primitives
- We can abstract methods to create new primitives

- Anything computable in X language will be in the Y one, but some language are better for special things (such as Matlab, to manipulate matrices.)

Quizz summary:

- A stored program computer is not designed to compute precisely one computation, such as a square root, or the trajectory of a missile.
- A fixed program computer is not designed to run any computation, by interpreting a sequence of program instructions that are read into it.
- A program counter points the computer to the next instruction to execute in the program
- The computer executes the instructions mostly in a linear sequence, except sometimes it jumps to a different place in the sequence “the computer walks through the sequence executing some computation"
- In order to compute everything that is computable, every computer must be able to handle the six most primitive operations (Right, Left, Print, Scan, Erase, Do Nothing; Turing Complete)

Language

Video summary:

- A programming language provides a set of primitive operations
- Expressions are complex but legal combinations of primitives in a language
- Expressions and computations have values and only one meaning.
- Primitive construct: numbers, strings, simple operators
- Static semantics such as 3+”hi” are an error
- Semantics: programming languages one meaning
- Syntactic errors: common and easily caught

Quizz summary:

- Syntax determines whether a string is legal
- Static Semantics determines whether a string has meaning
- Semantics assigns a meaning to a legal sentence

Types

Video summary:

- int; float; NoneType; bool
- type(number)
- python shell is an interpreter

Quizz summary:

- In Python, the keyword None is frequently used to represent the absence of a value. None is the only value in Python of type NoneType.
- Decimal numbers cannot be stored exactly in the computer because the computer does not have an infinite amount of memory. So decimal numbers are rounded when stored. When you do calculations with these numbers, your final result will be different than the actual result. For example, you may get something like 5.0000000044 instead of 5.0. This is called floating-point rounding error.

Variables

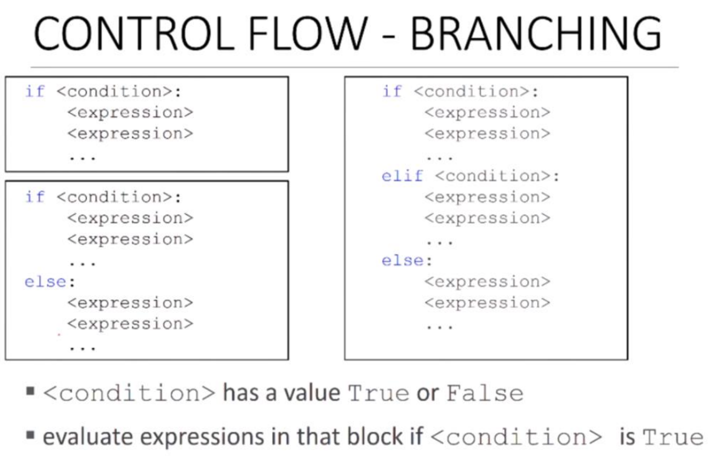
<u>Video summary:</u>

- Equal sign is an assignement of value to a variable name
- Value stored in computer memory; retrieve the value of a variable by invoking the name
- Give names to values of expressions to store it and avoid to redo the calculation
- Can re-bind variables names
- Previous value may still stored in memory but lost the handle for it

Operators and Branching

<u>Video summary:</u>

- Comparison operators on int and float: >; >=; <; <=; ==; !=
- Comparison operators on bool: not a; and; or
- The simplest branching statement is a conditional: a test; a block of code (if it's true); an optional block of code (if it's false)
- Drawing a branching program: rectangle = what to do; lozenge = question (if)
- Equivalent of branching program in programming: if/else
- Nested conditionals : conditions into conditions
- Compound Booleans : example: if x>y and z<x: need to check the two conditions are true to return true.



CONTROL FLOW - BRANCHING

```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

- `<condition>` has a value `True` or `False`
- evaluate expressions in that block if `<condition>` is `True`

-
- Identitation : how you denote blocks of code; really important in python. Aren't {} such C, only the syntax matters.

<u>Quizz summary:</u>

- Remember that in Python words are case-sensitive. The word True is a Python keyword (it is the value of the Boolean type) and is not the same as the word true. Refer to the Python documentation on Boolean values.

- For these problems, it's important to understand the priority of Boolean operations. The order of operations is as follows:

1. Parentheses. Before operating on anything else, Python must evaluate all parentheticals starting at the innermost level.
2. not statements.
3. and statements.

4. or statements.

What this means is that an expression like

```
not True and False
```

evaluates to False, because the not is evaluated first (not True is False), then the and is evaluated, yielding False and False which is False.

However the expression

```
not (True and False)
```

evaluates to True, because the expression inside the parentheses must be evaluated first - True and False is False. Next the not can be evaluated, yielding not False which is True.
Overall, you should always use parenthesis when writing expressions to make it clear what order you wish to have Python evaluate your expression. As we've seen here, not (True and False) is different from (not True) and False - but it's easy to see how Python will evaluate it when you use parentheses. A statement like not True and False can bring confusion!

B: Core Elements of Programs

Knowledge

Video summary:

- Variables, often called bindings these were names that we had to which we could associate values.
- Name = value; name can be descriptive, meaningful, helpful to re-read code, cannot be keywords (eg int or float word)
- Value are information stored and can be updated.
- Swap variable using temp variable to store a variable and swap between two.

String

Video summary:

- Strings are sequences of characters, enclosed in quotation mark or single quote
- “a”+”b” = concatenation; 3*”a” = successive concatenation; len(“a”) = length; “aacba”[1:3] = sclicing (return “ac”) (‘moka’[:] will return “moka”)

Quizz summary:

- you can slice a string with a call such as s[i:j], which gives you a portion of string s from index i to index j-1.
- If you omit the starting index, Python will assume that you wish to start your slice at index 0. If you omit the ending index, Python will assume you wish to end your slice at the end of the string.
- s[i:j:k]. This gives a slice of the string s from index i to index j-1, with step size k.

```
- >>> s = 'Python is Fun!'
- >>> s[1:12:2]
- 'yhni u'
```

- Str1 = “moka” str1[-1] will return a.

Input/Output

Video summary:

- print to output stuff to console: print(x,”z”,y) || print(x + y + “z”)
- input to input stuff: text = input(“type something:”); I type “moka”; text; “moka”
- input return a string so must cast if working with numbers: int(input(day:))
-

IDE’s

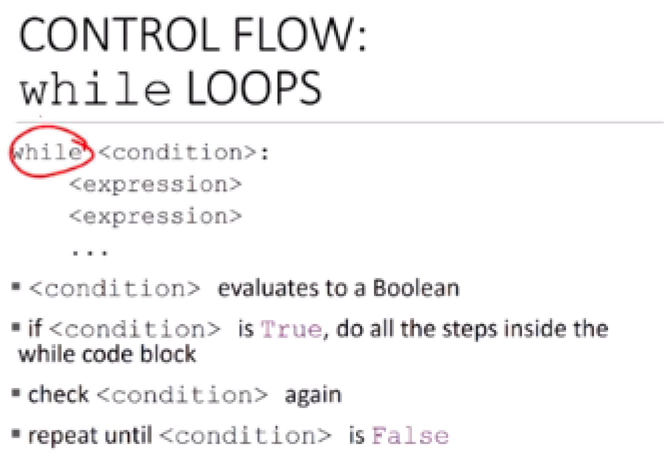
Video summary:

- IDE = integrated development environment
- Have a text editor (enter,edit,save programms), a shell and an integrated debugger

Control flows

Video summary:

- Remind: The simplest branching statement is a conditional.
- Simple branching programs just make choices != something that can be reused.
- Using control in loops: reuse parts of the code indeterminate number of times.
- Keyword: while <conditions> : or for loop.



CONTROL FLOW:
while LOOPS

```
while <condition>:
    <expression>
    <expression>
    ...
```

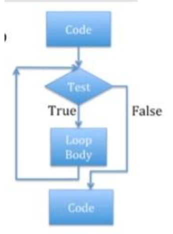
- <condition> evaluates to a Boolean
- if <condition> is True, do all the steps inside the while code block
- check <condition> again
- repeat until <condition> is False

- Loop continue until it's false.
- For loop: for <variable> in <expression>: (eg: for x in range(5*):) *range of n will go to 0 to n-1.
- range(start,stop,step).
- break statement exit the loop, skipping the remaining conditions of this loop. Only exist innermost loop.

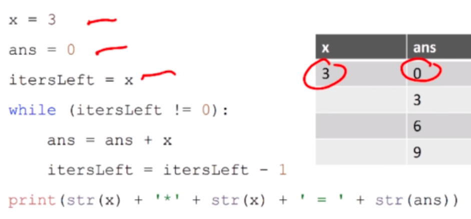
Iteration

Video summary:

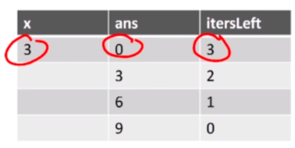
- Concept of iteration lets us extend simple branching algorithm to be able to write programs of arbitrary complexity.
- Start with a test, if evaluates to True, execute loop once and go back to test. If false skip the loop and go back to the code.



-
- Need to set an iteration variable outside the loop
- Need to test variable outside the loop
- Need to change variable within the loop, in addition to other work



```
x = 3
ans = 0
itersLeft = x
while (itersLeft != 0):
    ans = ans + x
    itersLeft = itersLeft - 1
print(str(x) + '*' + str(x) + ' = ' + str(ans))
```



x	ans	itersLeft
3	0	3
	3	2
	6	1
	9	0

Guess and Check

Video summary:

- Iterative algorithms allow us to do more complex things than simple arithmetic.
- We can repeat a sequence of steps multiple times based on some decision; leads to new classes of algorithms.
- "guess and check" methods: example: algorithm to find a perfect square root from x number, stop if k**n > x
- Extending scope as a method of guess only works for positive integers and are easy to fix keeping track of sign, looking for solution to positive case (eg. Use the abs() func)
- Decrementing function: when loop is entered, value is non-negative; when value is <= 0, loop terminates; value is decreased every time through loop
- Process of exhaustive enumeration: able to guess a value; able to check if the solution is correct; keep guessing until find a solution or guessed all values.

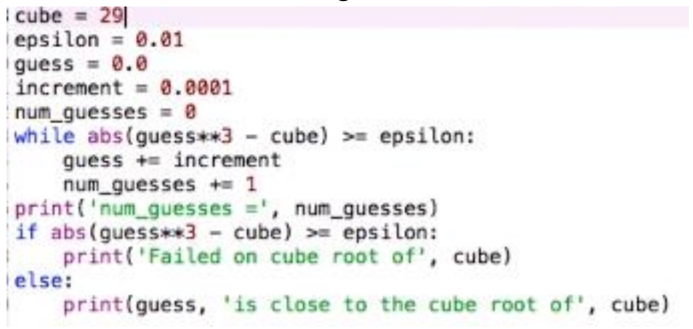
Week 2: Simple Programs

A: Simple Algorithms

Approximate Solutions

<u>Video summary:</u>

- Example: exhaustive enumeration -> a good enough solution: |guess**3|-cube <= epsilon; decrease/increment size of guess will be slower BUT increasing epsilon will be less accurate.



```
cube = 29
epsilon = 0.01
guess = 0.0
increment = 0.0001
num_guesses = 0
while abs(guess**3 - cube) >= epsilon:
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**3 - cube) >= epsilon:
    print('Failed on cube root of', cube)
else:
    print(guess, 'is close to the cube root of', cube)
```

-
- Need to make sure that the guess isn't too big !
- Small step will be more accurate but slower, if it's too large it may skip the answer.

Bisection Search

<u>Video summary:</u>

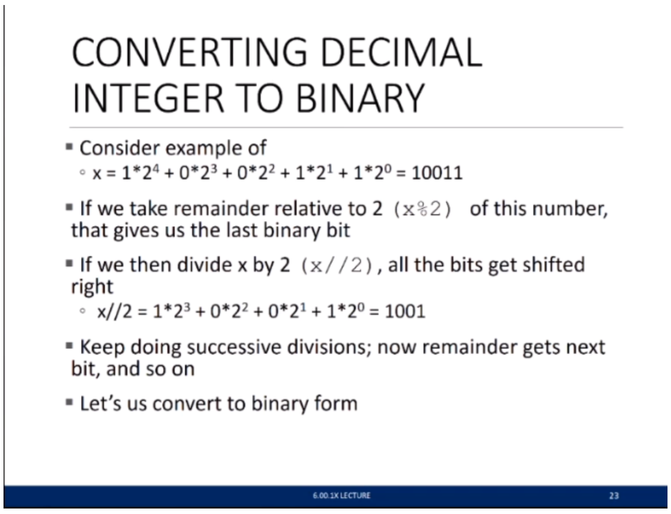
- Rather than exhaustively trying thigs starting at "A" (like exhaustive enumeration), we can pick a number in the middle of this range.
- If not close enough? Is guess too big or too small? Redo in the biggest or smallest range and redo…
- Bisection search convergence: Xth guess = var/2**X; Guess converges on the order of log_2_N steps
- Works when value of function varies monotonically w/ input
- This search method radically reduces computation time
- Work well on problems with ordering property. (square root of x grows as x growns)
- Add the following ,end='-' in a print statement will make print the next print next to the text between ''.

<u>Test:</u>

```
#find number between [0;100[
lower = 0
upper = 100
find = False
secret = "a"
print("Please think of a number between 0 and 100!")
while find == False:
   guess = (upper + lower)//2
   print("Is your secret number "+ str(guess) + "?")
   secret = input("Enter 'h' to indicate the guess is too high. Enter 'l' to indicate the guess is too low.
Enter 'c'")
   if secret == "h":
        upper = guess
    elif secret == "l":
        lower = guess
    elif secret == "c":
        break
  else:
      print("Sorry, I did not understand your input.")
print("Game over. Your secret number was: " + str(guess))
```

Floats and Fractions:

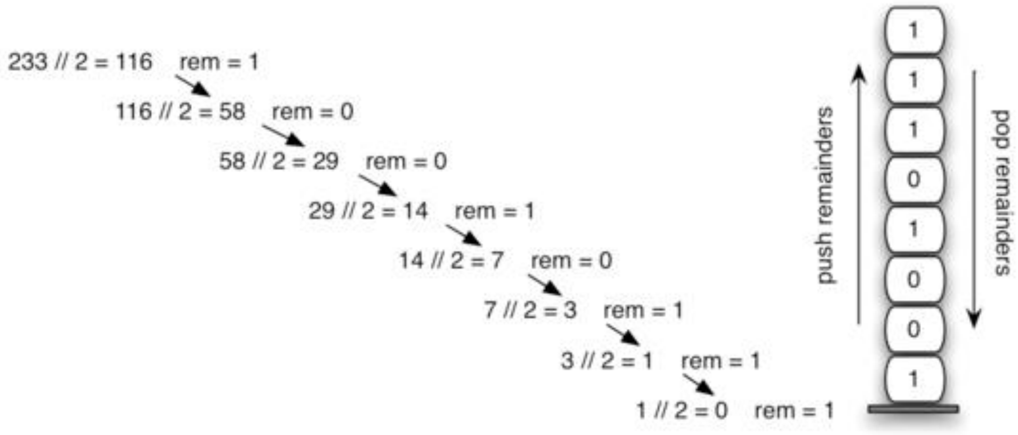
<u>Video summary:</u>

- decompouse: xyz = x*(base)**2 + y*(base)**1 + z*(base)**0
- decompouse 0,xyz = 0*(base)**0 + x*(base)**-1 + y*(base)**-2 + z*(base)**-3
- 

CONVERTING DECIMAL INTEGER TO BINARY

- Consider example of
  - $x = 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 10011$
- If we take remainder relative to 2 (`x%2`) of this number, that gives us the last binary bit
- If we then divide x by 2 (`x//2`), all the bits get shifted right
  - $x//2 = 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 1001$
- Keep doing successive divisions; now remainder gets next bit, and so on
- Let's us convert to binary form

6.00.1X LECTURE 23

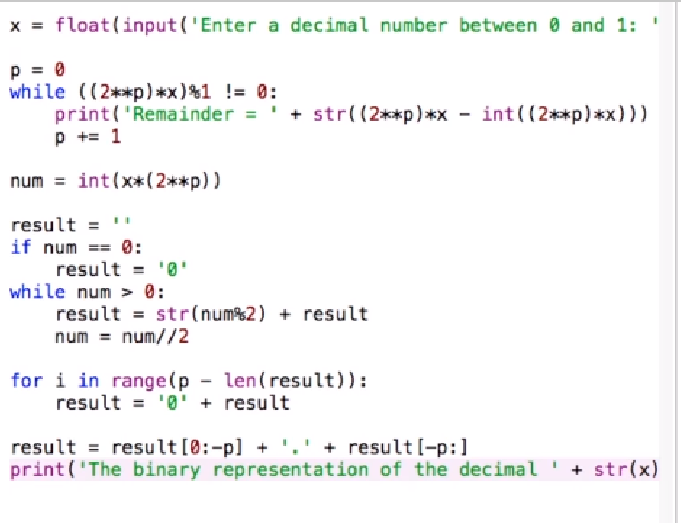


-
- Trick: multiply the float by a power of 2 big enough to convert it to a integer, just need to convert this number in binary then divide by the same power of 2.

```
x = 25
count = 0

result = ''
if x < 0:
    neg = True
```

```
else:
   neg = False
while x > 0:
   result = str(x%2) + result
   x = x//2
if neg:
   result = '-' + result
```



```
x = float(input('Enter a decimal number between 0 and 1: '

p = 0
while ((2**p)*x)%1 != 0:
    print('Remainder = ' + str((2**p)*x - int((2**p)*x)))
    p += 1

num = int(x*(2**p))

result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2

for i in range(p - len(result)):
    result = '0' + result

result = result[0:-p] + '.' + result[-p:]
print('The binary representation of the decimal ' + str(x)
```

- Suggest that testing equality of floats: use abs(x-y) < some small number; rather than x == y

Newton-Raphson:

<u>Video summary:</u>

- General approximation algorithm to find roots of a polynomial in one variable:
  p(x) = $A_n X^n + A_{n-1} X^{n-1} + \cdots + A_1 X^0$
- Newton showed that if g is an approximation to the root, g-p(g)/p’(g) is a better approximation; where p(x) = x^2 - a, where a is the number which need to find its square root.

B: Functions

Decomposition and Abstraction:

<u>Video summary:</u>

- Functions: mechanism to achieve decomposition and abstraction; a way to encapsulate pieces of information
- Abstraction idea: once I've built something, I don't need to know what's inside it as long as I know how it works (ex. projector: if you give me appropriate inputs, going to behave in an appropriate way)
  In Programming: suppress details of method to compute something from use of that computation
- Decomposition idea: different devices work together to achieve an end goal (ex. Separate projectors: each one takes input and produce separate output, they all work together to produce larger image)
  In Programming: break problem into different, self-contained, pieces

- In programming, abstraction consist to think of a piece of code as a black box
  - cannot see details
  - do not need/want to see details
- In programming, decomposition consist in divide code into modules:
  - are self-contained
  - used to break up code
  - intended to be reusable
  - keep code organized
  - keep code coherent
- Decomposition & abstraction are powerful together

Introducing Functions:

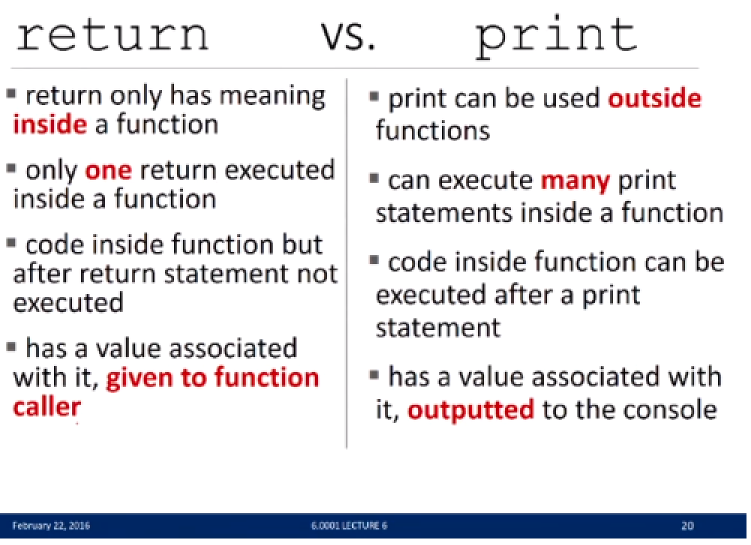
<u>Video summary:</u>

- Functions are not run in a program until they have called or invoked in a program
- Functions has a name, parameters(0 or more), docstring (optional), body
- Keyword: def.
  def function_name(argument):
- Keyword docstring : “”” informations “””
- To call a function: function_name(an_argument)
- Return value with return

Calling Functions and Scope:

<u>Video summary:</u>

- Formal parameter gets bound to the value of actual parameter when function is called
- New scope/frame/environment created when enter a function
- Scope is mapping of names to objects
- If no return statement: python will return none, absence of value



return vs. print

return	print
▪ return only has meaning **inside** a function	▪ print can be used **outside** functions
▪ only **one** return executed inside a function	▪ can execute **many** print statements inside a function
▪ code inside function but after return statement not executed	▪ code inside function can be executed after a print statement
▪ has a value associated with it, **given to function caller**	▪ has a value associated with it, **outputted** to the console

February 22, 2016 6.0001 LECTURE 6 20

- Arguments can take on any type, even functions
- Inside a function, can access a variable defined outside
- Inside a function, cannot modify a variable defined outside
- Cannot modify a variable if no return at the end of the function

Keyword arguments & Specification:

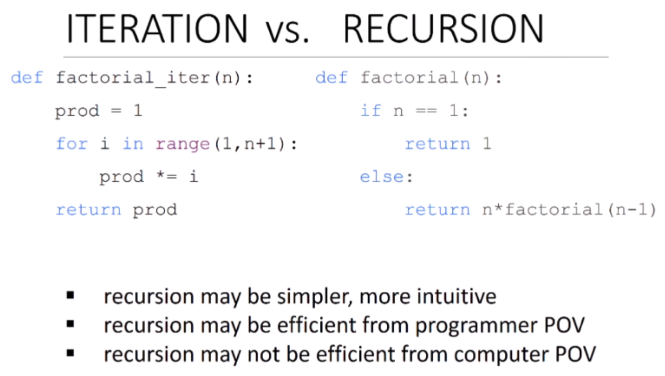
<u>Video summary:</u>

- If we precise argument in the call, we can forgot the order defined in the function.
  def name(first, last); name(“mo”, “ka”) == name(first=”mo”, last=”ka”)
- Specifications: assumptions: conditions that must be met by clients of the function; typically constraints on value of parameters
- Specification: Guarantees: conditions that must be met by function, providing it has been called in manner consistent with assumptions
- Assumptions and guarantees are written in docstring

Iteration vs Recursion:

<u>Video summary:</u>

- Recursion is a way to design solutions to problems by divide-and-conquer or decrease-and-conquer
- A programming technique where a function calls itself
- In programming, goal is to NOT have infinite recursion.
- Must have one or more base cases that are easy to solve
- Must solve the same problem on some other input with the goal of simplify the larger problem input
- Multiplication in ITERATIVE solution: multiply a*b is equivalent to add a to itself b times
- Multiplication in RECURSIVE solution:
  - Recursive step: think how to reduce problem to a simpler/smaller of same problem
    (a*b = a + a * (b-1)) -> return a + function_name(a, b-1)
  - Base case: keep reducing problem until reach a simple case that can be solved directly
    (when b=1, a*b = a)
- Same way for factorial: return n*function_name(n-1)
- Each recursive call to a function creates its own scope
- Bindings of variables in a scope is not changed by recursive call
- Flow of control passes back to previous scope once function call returns value



ITERATION vs. RECURSION

```
def factorial_iter(n):
    prod = 1
    for i in range(1,n+1):
        prod *= i
    return prod
```

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

- recursion may be simpler, more intuitive
- recursion may be efficient from programmer POV
- recursion may not be efficient from computer POV

-

Inductive Reasoning:

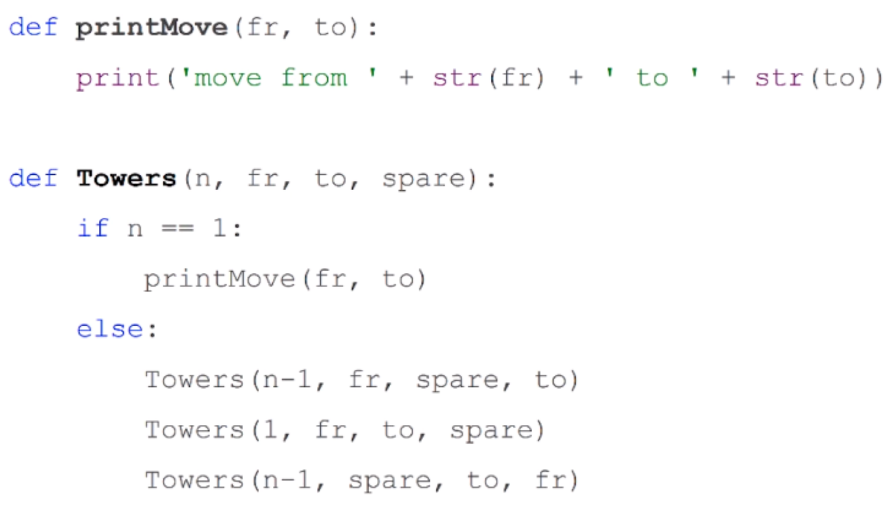
<u>Video summary:</u>

- When writing a recursive function, be sure to change the parameter to avoid infinite loop
- Mathematical Induction: to prove a statement indexed on integers is true for all values of n:
  - Prove it when n is smallest value
  - Then prove that if it is true for an arbitrary value of n, one can show that it must be true for n+1

Towers of Hanoi:

<u>Video summary:</u>

- Basic: we have 3 tall spikes, and we need to stack 64 different sized discs. Start on one spike, need to move to stack to second spike (at which point universe ends) and can only move one disc at a time, and a larger disc can never cover up a small disc.
- Linked with recursion:
  - Solve a smaller problem
  - Solve a basic problem
  - solve a smaller problem,…



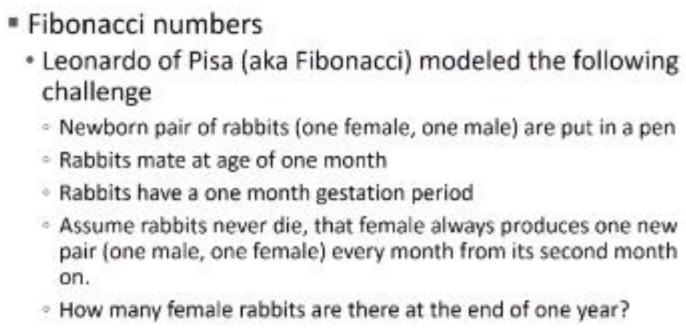
```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))


def Towers(n, fr, to, spare):
    if n == 1:
        printMove(fr, to)
    else:
        Towers(n-1, fr, spare, to)
        Towers(1, fr, to, spare)
        Towers(n-1, spare, to, fr)
```

-

Fibonacci:

<u>Video summary:</u>



- Fibonacci numbers
  - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
    - Newborn pair of rabbits (one female, one male) are put in a pen
    - Rabbits mate at age of one month
    - Rabbits have a one month gestation period
    - Assume rabbits never die, that female always produces one new pair (one male, one female) every month from its second month on.
    - How many female rabbits are there at the end of one year?

-
- In month (0) 1 female; second month (1) 1 female (P); third month (2) 2 female, one (P)…
- Recursive case: females(n) = females(n-1) + females(n-2)
- Recursive cases with many bases cases

Non-numerics Recursion:

<u>Video summary:</u>

- How to check if a string is a palindrome?
- Eg: able was I ere I saw Elba
- How to do: first convert to just characters.
  - base case: len 0 = empty; len 1 palindrome
  - how to check in recursive: check if n and –n are the same letter, if yes, delete n and –n do the same check until the end.

Modules and Files:

Video summary:

- Cumbersome for large collections of code, or for code that should be used by many different other pieces of programming
- A module is a .py file containing a collection Python definitions and statements
- We can import module with keyword import module_name
- To use a function of the module: module_name.module_function
- from module_name import * -> import everything from the module
- from module_name import function -> import the function from the module
- Every operationg system has its own way of handling files; Python provides an operationg-system independent means to access files, using a file handle.
- nameHandle = open(‘a’, ‘w’) -> create a file named a and return the file handle which we can name thus reference. ‘w’ indicates that the files is to opened for writing into.
- Other keyword:
  - filename.write(contents) -> write in the file
  - filename.close() -> close the file
- Filename.open(`file`, ‘r’) -> to read the content of the file