Python DeCal

Week 3

Announcements

- 1st Hw is due on Wednesday + Resource notebook available
- Attendance!
  - https://forms.gle/E3gkGF2r561pvHX3A
- Office Hours scheduled

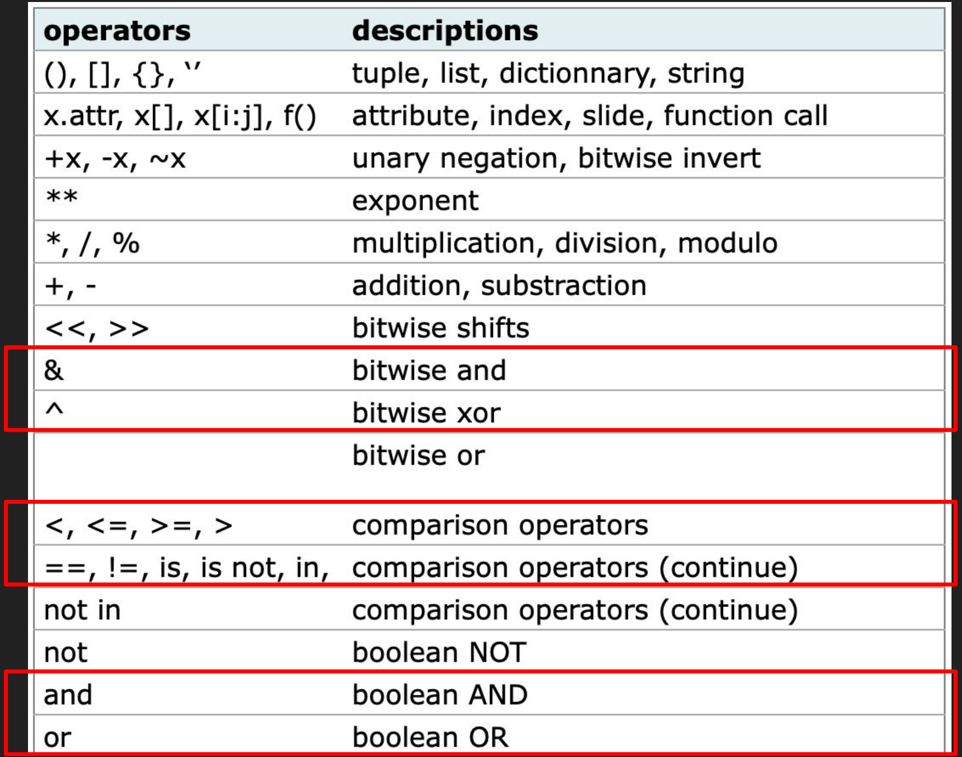
Recap

- Data Type (int, float, string, lists, dictionary, tuples, booleans)
- Functions
  - Construct a function
  - Calling a function
- Variable scope

However...

- Our code can’t make any “judgements”
- It can’t change the flow of the code according to the input/variable

Boolean Operations



operators	descriptions
(), [], {}, ''	tuple, list, dictionnary, string
x.attr, x[], x[i:j], f()	attribute, index, slide, function call
+x, -x, ~x	unary negation, bitwise invert
**	exponent
*, /, %	multiplication, division, modulo
+, -	addition, substraction
<<, >>	bitwise shifts
&	bitwise and
^	bitwise xor
	bitwise or
<, <=, >=, >	comparison operators
==, !=, is, is not, in,	comparison operators (continue)
not in	comparison operators (continue)
not	boolean NOT
and	boolean AND
or	boolean OR

→ not the normal and

→ not exponent

```
_______ and/or _______
```

and:

- Return True if both True
- Return False if one is False

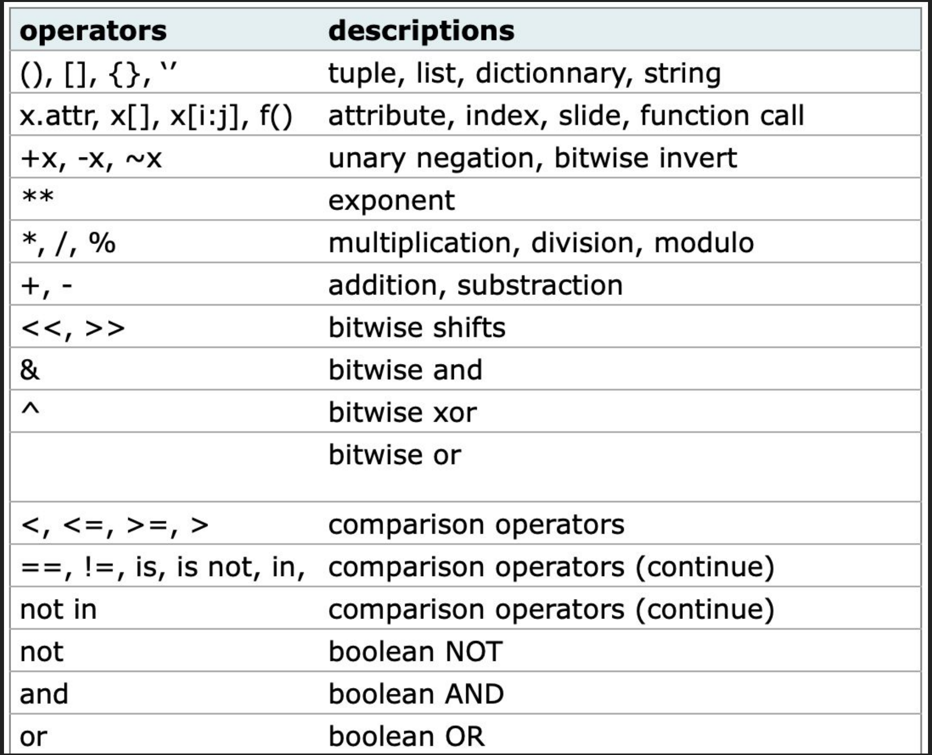
or:

- Return True if one is True
- Return False if both are False

Boolean Operations

- True or False?

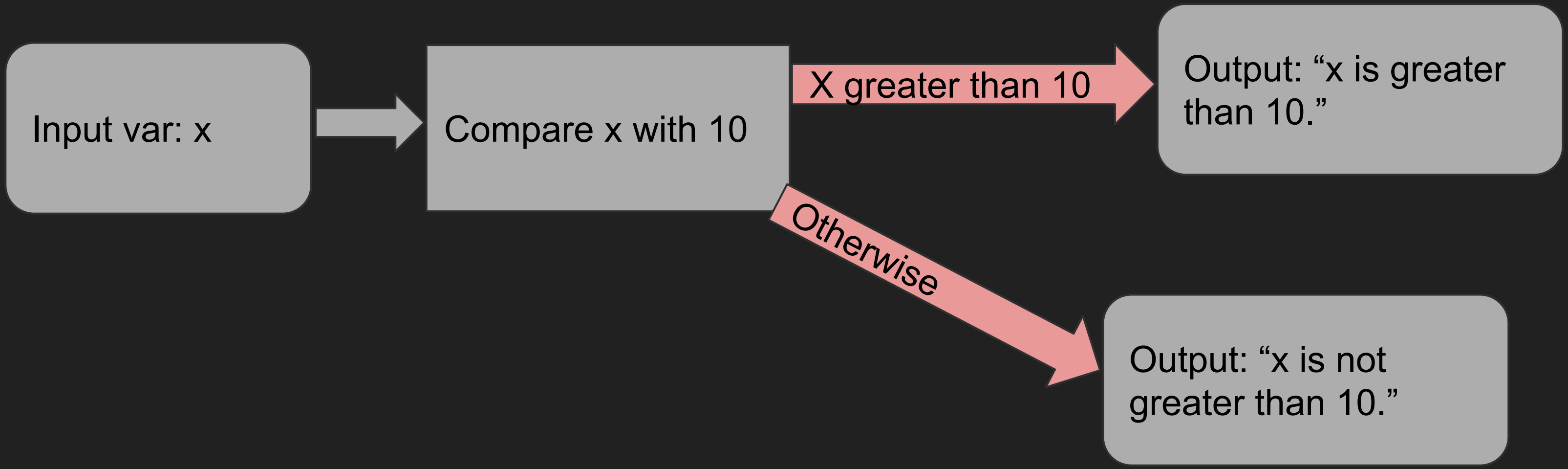
1. True and True → True
2. True or False → True
3. 1 >= 2 → False
4. 5%2 == 1 → True
5. (2!=2) or (1==1) → True
6. 2!=2 or 1==1 → True
7. 3/0 or True → Error
8. True or 3/0 → True
9. False and 3/0 → False
10. not 2==1 → True



operators	**descriptions**
(), [], {}, ‘’	tuple, list, dictionnary, string
x.attr, x[], x[i:j], f()	attribute, index, slide, function call
+x, -x, ~x	unary negation, bitwise invert
**	exponent
*, /, %	multiplication, division, modulo
+, -	addition, substraction
<<, >>	bitwise shifts
&	bitwise and
^	bitwise xor
	bitwise or
<, <=, >=, >	comparison operators
==, !=, is, is not, in,	comparison operators (continue)
not in	comparison operators (continue)
not	boolean NOT
and	boolean AND
or	boolean OR

If Statements

- Write a piece of code that outputs if the variable is greater than 10:


Input var: x
Compare x with 10
X greater than 10
Output: “x is greater than 10.”
Otherwise
Output: “x is not greater than 10.”

If Statements

- Write a piece of code that outputs if the variable is greater than 10:

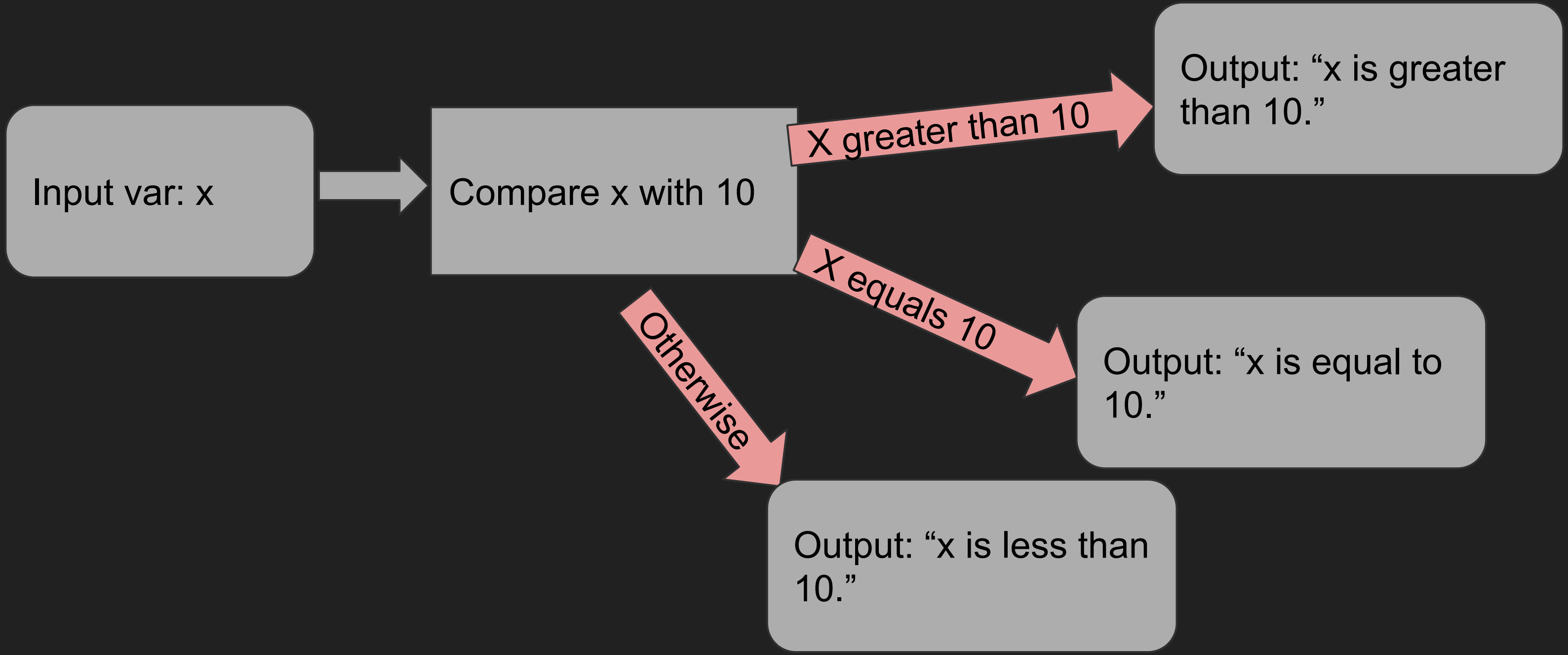
```
x = int(input(“please enter an integer:”))
if x > 10:
    print(“x is greater than 10.”)
else:
    print(“x is not greater than 10.”)
```

Ben Porter
@eigenbom

I'll sometimes leave a dangling else just as a threat to the

If Statements

- What if also we want to separate the equal case?


Input var: x
Compare x with 10
X greater than 10
Output: “x is greater than 10.”
X equals 10
Output: “x is equal to 10.”
Otherwise
Output: “x is less than 10.”

If Statements

- Write a piece of code that outputs if the variable is greater than 10:

```
x = int(input(“please enter an integer:”))
if x>10:
    print(“x is greater than 10.”)
elif x==10:
    print(“x is equal to 10”)
else:
    print(“x is smaller than 10.”)
```

Else if statement

If Statements

- We can also write nested if statements:
- We want to find if x can be divided by 2 and 3:

```
x = int(input(“please enter an integer:”))
if x%2 == 0:
    if x%3 == 0:
      print(“2 and 3 both divides x.”)
else:
    print(“2 and 3 can’t divide x simultaneously.”)
```

- Can you write it without the nested statements?

If Statements

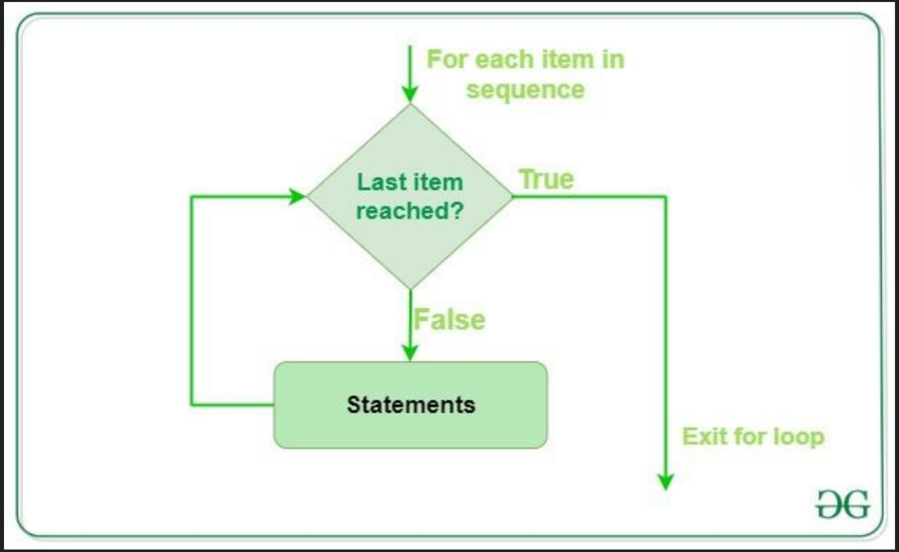
- Challenge: now we want to include all the consequences--
    - Can be divided by 2 but not by 3
    - Can be divided by 3 but not by 2
    - Can’t be divided by 3 nor 2

```
x = int(input(“please enter an integer:”))
if x%2 == 0 and x%3 == 0:
    print(‘Divisible by 2 and 3.’)
elif x%2 == 0:
    print(‘Divisible by 2.’)
elif x%3 == 0:
    print(‘Divisible by 3.’)
else:
    print(‘Not divisible by 2 or 3.’)
```

Iterative tasks

- Now we want the code to run n times
- For example, you wanna spam your friends (don’t do that). You would want the code to run numerous times (like 100 times *precisely*).
- If statements do not do that :c

Loops-for loop



```
for item in list:
    do something
```

Loops-for loop

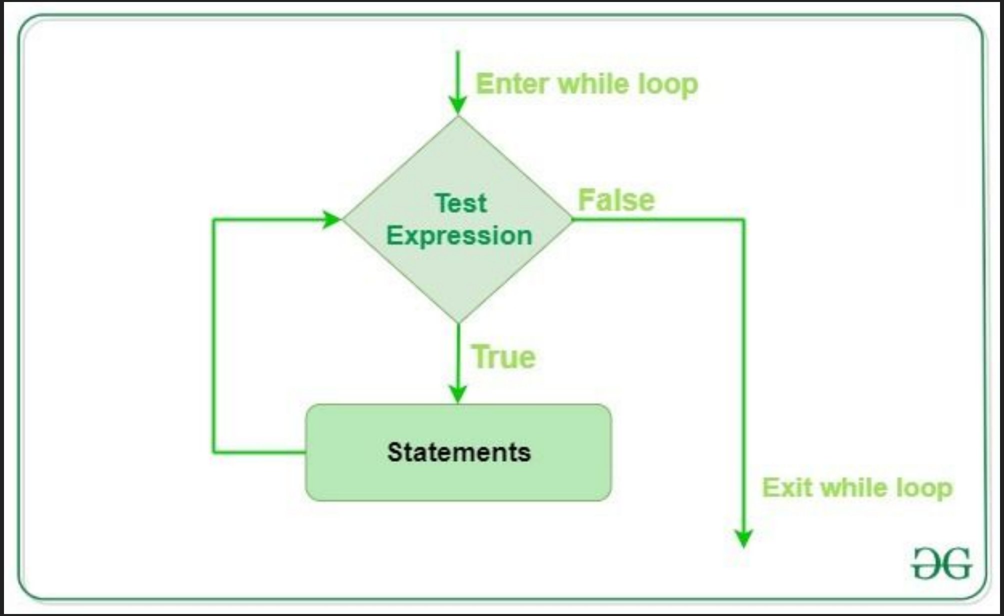
- For loop: iterate over items in a list
- Then we need either run commands on existing lists, or create new list for it to run over.
- For example, you want to print numbers from 0 to 100. It would be a long task for you if you wanted write them all out!

Loops-for loop

- Print numbers from 1 to 100.

```
for i in range(1,101):
    print(i)
```

Loops-while loop



```
while value is True:
    do something
```

Loops-while loop

- While loop: print numbers 1 to 5

```
i = 1
while i < 6:
    print(i)
    i += 1
```

- What will be the output now?

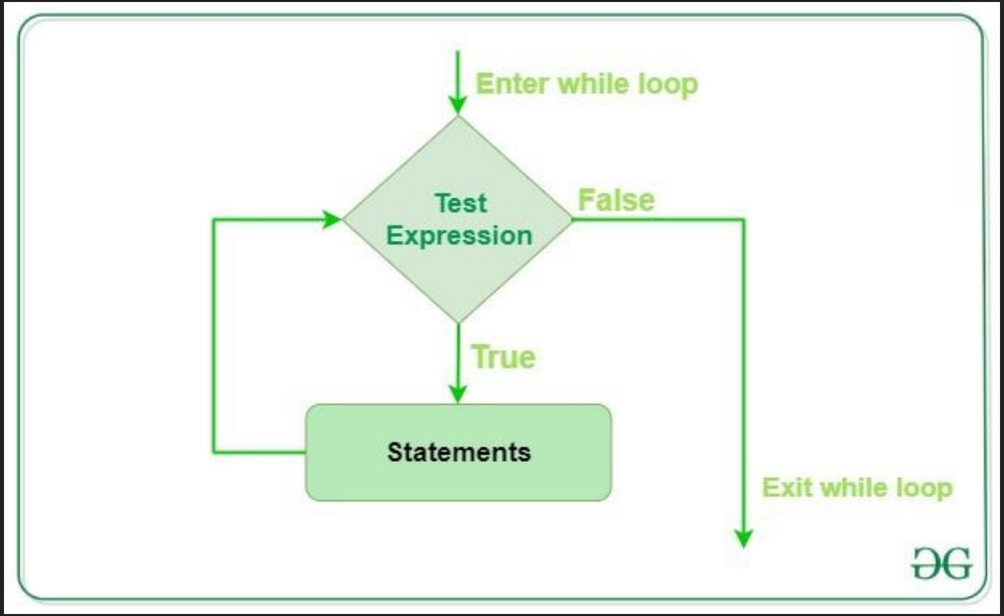
```
i = 1
while i < 6:
    i += 1
    print(i)
```

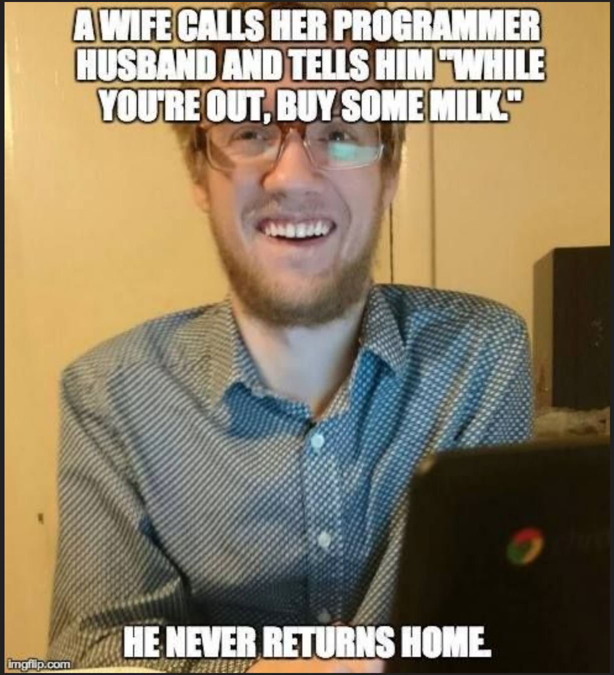
Loops-while loop

- What will be the output now?

```
i = 1
while i < 6:
    print(i)
```

Loops-while loop



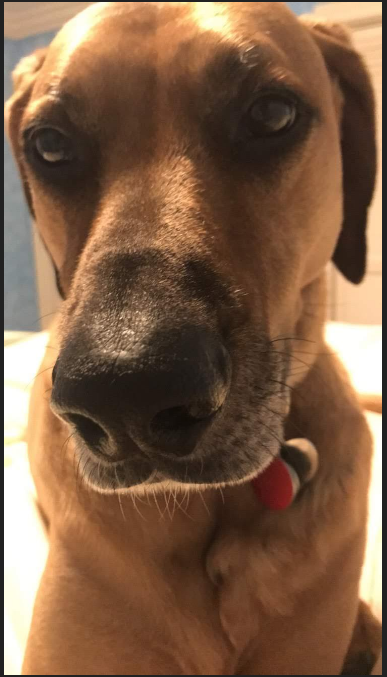

A WIFE CALLS HER PROGRAMMER HUSBAND AND TELLS HIM "WHILE YOU'RE OUT, BUY SOME MILK."
HE NEVER RETURNS HOME.
imgflip.com

Loops-while loop

```
out = True
milk = 0
while out == True:
    print(“Buy milk”)
    milk += 1
    if milk > 0:
        break
print(“The programmer is back home”)
```

Demo

WEDNESDAY



Announcements

- Homework 2 will be up soon. It will take a bit more time than last week so be prepared.
- Office Hours: Check bCourses, same Zoom link as lecture
- Today: dictionaries and recursion

Attendance Form

- https://forms.gle/yXLpPSFHn5JcCJmE6

If Statements

- Write a piece of code that outputs if the variable is greater than 10:

```
x = int(input(“please enter an integer:”))
if x>10:
    print(“x is greater than 10.”)
elif x==10:
    print(“x is equal to 10”)
else:
    print(“x is smaller than 10.”)
```

Else if statement

Break Out Rooms!

Question: What is the difference between a `while` and a `for` loop and why would you want to use each of them?

While

- Use when you want something to happen as long as condition is met

```
While (condition is True):
    Do some stuff
    Something will change each time
```

Prevents infinite loops!

For

- Used when you want to iterate over something (a list, a tuple, etc...) but you don't necessarily need a condition to be met while it does this.

```
For (thing in a bunch of things):
    Do some stuff for each thing in a bunch of things
```

Examples

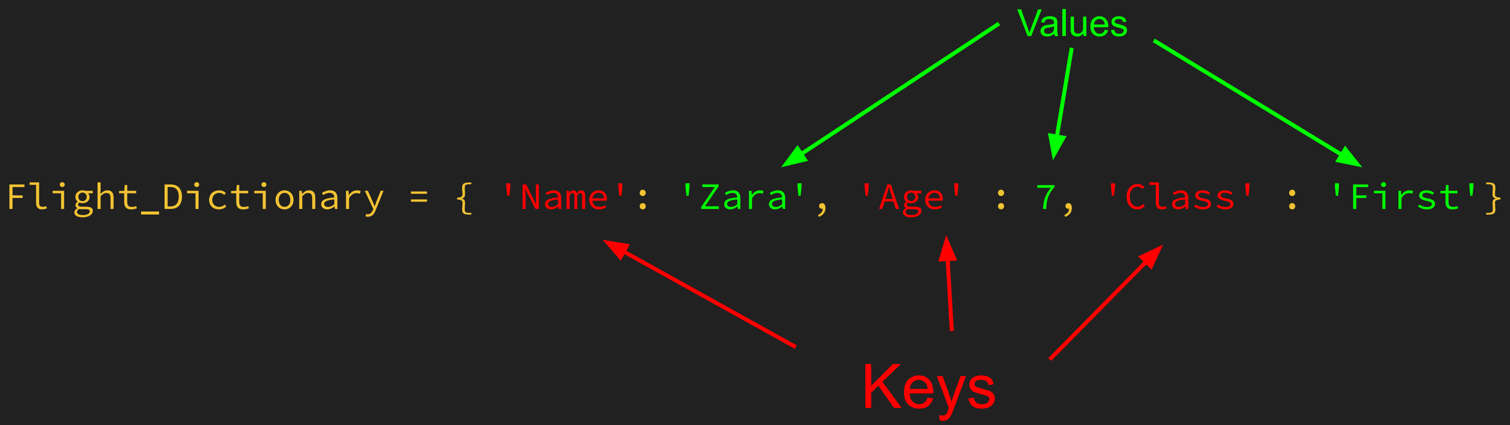
`While`

- Video Games
- Password Inquirer
- Simulations
- Steps don’t need to be Discrete integer values

`For`

- Data Sets (large lists or files)
- Grid Simulations
- Changing specific values in a list
- Discrete integer steps

Dictionaries { }: The last built-in data type of python!



```
Flight_Dictionary = { 'Name': 'Zara', 'Age' : 7, 'Class' : 'First'}
```

- Dictionaries are (key, value) pairs that do not have order
- Keys: index of a dictionary
  - Think of looking up a word (key) in a dictionary
- Values: data associated with a certain index
  - Think of like the definition (value) of a word in a dictionary

Dictionaries

How to create and fill a dictionary:

1) d = {} or d = dict(), d[key] = value

```
>>> Flight_Dictionary = {} or Flight_Dictionary = dict()
>>> Flight_Dictionary[‘Name’] = ‘Zara’ // Adds {‘Name’: ‘Zara’} to the dict
>>> Flight_Dictionary[‘Age’] = 7
```

2) d = {key1:value1, key2:value2} or d = dict([(key1,value1), (key2,value2)])

```
>>> Flight_Dictionary = {'Name': 'Zara', 'Age' : 7, 'Class' : 'First'}
                                      or
>>> Flight_Dictionary = dict([('Name', 'Zara'), (‘Age’, 7), (‘Class’, ‘First’)])
```

Dictionaries

Accessing values from keys

- d[key] or d.get(key)

```
>>> Flight_Dictionary[‘Name’]
    ‘Zara’
>>> Flight_Dictionary.get(‘Age’)
    7
```

Deleting keys

- del d[key]

```
>>> del Flight_dictionary[‘Name’]
    (deletes the (‘Name’: ‘Zara’) pair from Flight_dictionary)
```

Dictionaries

Accessing ALL values

- d.values()

```
>>> Flight_Dictionary.values()
    ['Zara', 7, 'First']
```

Accessing ALL keys

- d.keys()

```
>>> Flight_dictionary.keys()
    ['Name', 'Age', 'Class']
```

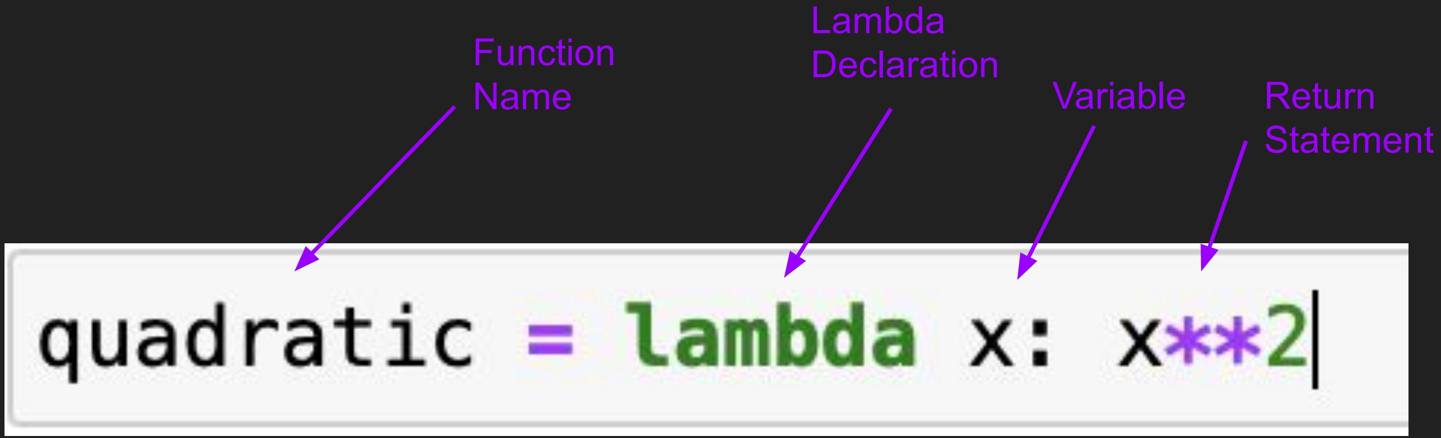
Dictionaries

Example:

```
SN1987a = { ‘Apparent Magnitude’:2.9,
            ‘Distance’:51.4,
            ‘RA’:‘05h 35m 28.03s’,
            ‘Dec’:‘-69° 16′ 11.79″’,
            ‘Type’:‘II’ }
```

Advanced/Extra Material

- Lambda functions!
- The quick and dirty way of making a function



```python
quadratic = lambda x: x**2
```

Recursion

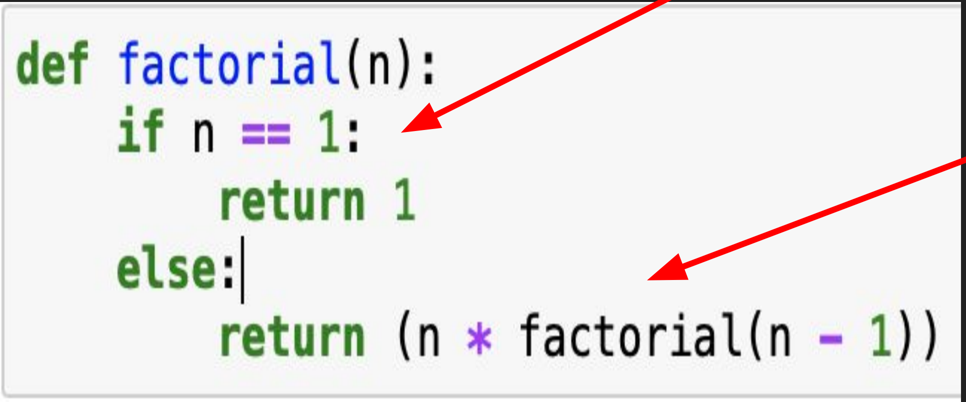
- Complicated topic, should take CS61A if you are really curious about it
- Essence… call a function on itself repeatedly to make it do what you want

$$X = \sqrt{6 + \sqrt{6 + \sqrt{6 + \sqrt{6 + \ldots\ldots}}}}$$

WE CAN CODE THIS….

Recursion

- How do we implement it ???
- Take a recursive leap of faith…
    - Trust that the recursive call will meet the base case eventually and return what you want



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

Recursion

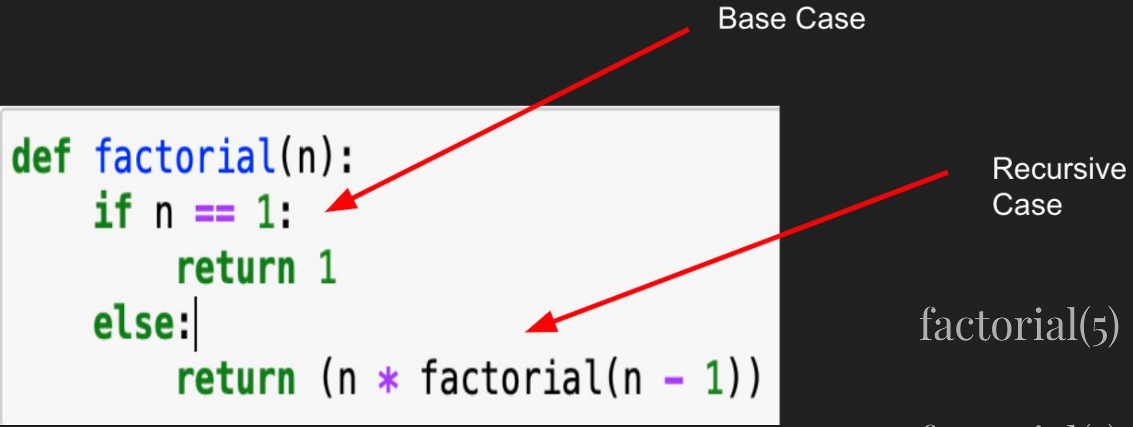
- Instead of returning a value (like 10, True, 'hi'), **return a call to the same function!**
- You should always have two groups of cases. 1. base case and 2. recursive case

- **Recursive case** is what is returned at the end of the function

  ⤐ Make sure to modify the arguments slightly (+, - , /, *) to avoid an infinite loop

- **Base case** is at the end of the long chain, which signals to stop the recursive calls and and start returning the previously calls
  ⤐ Think of the case when your function is the smallest value

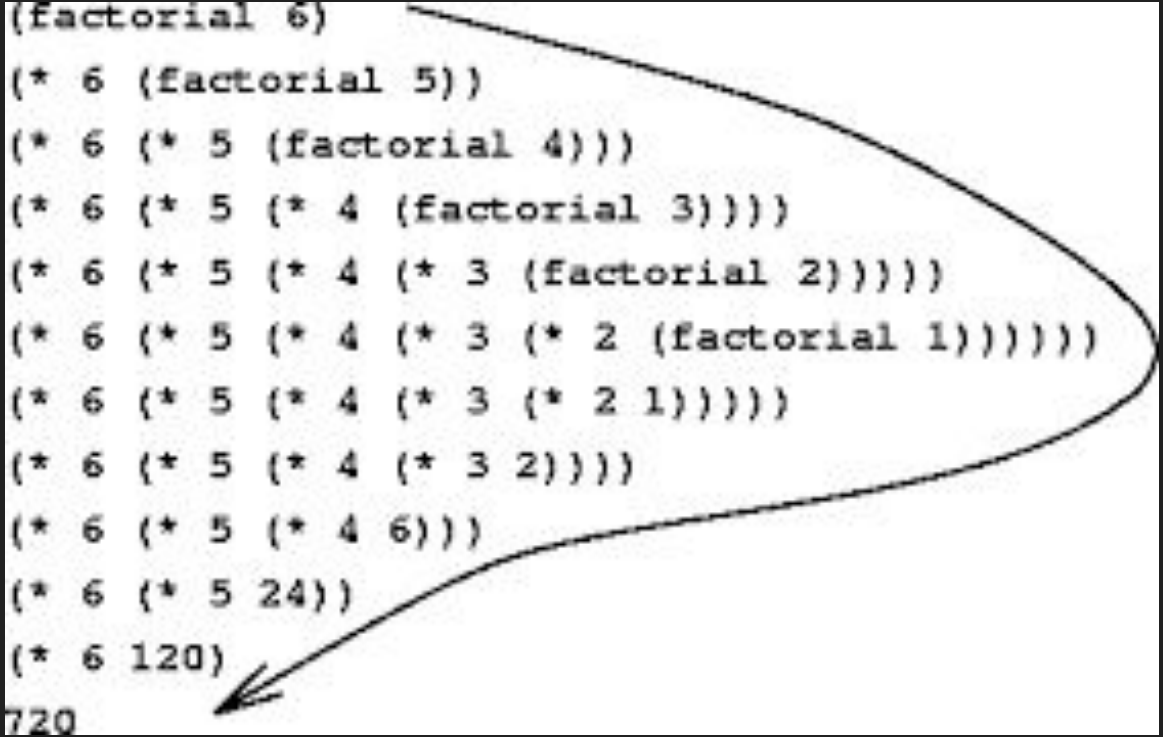
CARTOONSTOCK
Search ID: gra100910
9-10
guy & rodd
© Guy & Rodd/Distributed by Universal Uclick for UFS via CartoonStock.com



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

factorial(5) = 5*4*3*2*1

factorial(5)
5 * factorial(4)
5 * (4 * factorial(3))
5 * 4 * 3 * factorial(2)
5 * 4 * 3 * 2 * factorial(1)
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120

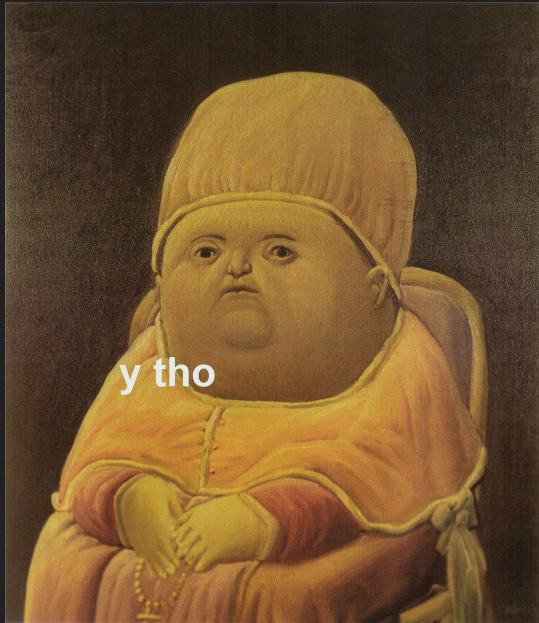


```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

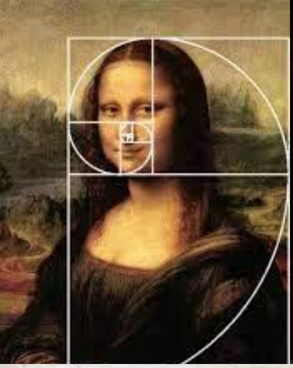
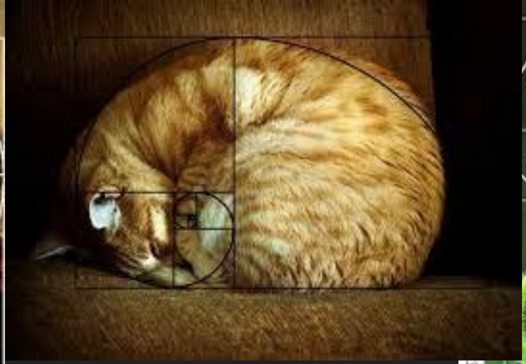
Note that in this diagram (* 2 1) just means (2 * 1)

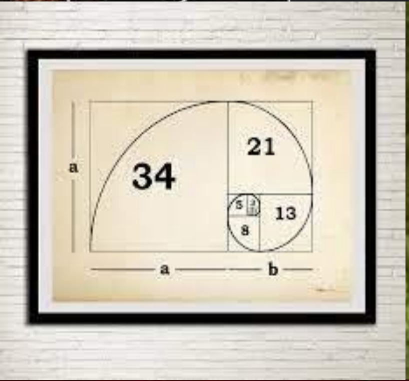
Why Recursion?

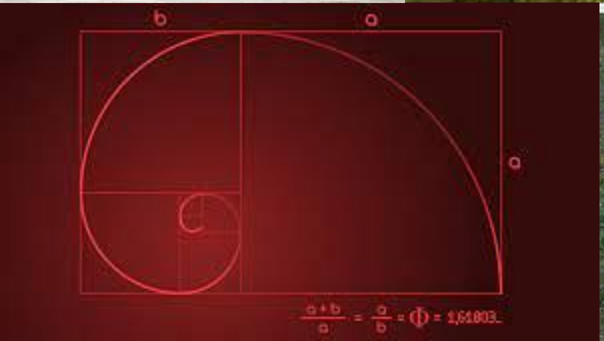
- Some ideas are better expressed as small steps
- Tree recursion to express multiple possible paths
- Can make code much cleaner and efficient

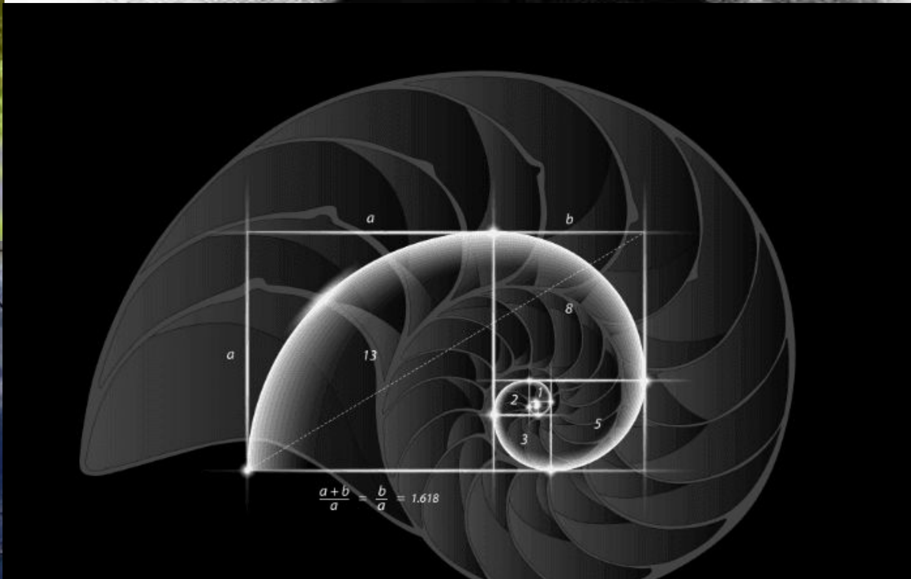


Demos


a
34
21
5
8
13
a
b


b
a
a
$\frac{a+b}{a} = \frac{a}{b} = \Phi = 1,61803...$


a
b
a
13
8
2
1
5
3
$\frac{a+b}{a} = \frac{b}{a} = 1.618$

Example: Fibonacci

WHILE METHOD:

```
def while_fib(n):
    fib1 = 0
    fib2 = 1
    k = 1
    while k < n:
        curr = fib1 + fib2
        fib1 = fib2
        fib2 = curr
        k = k + 1
    return curr
```

RECURSIVE METHOD:

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

IT'S PRETTY....

Questions