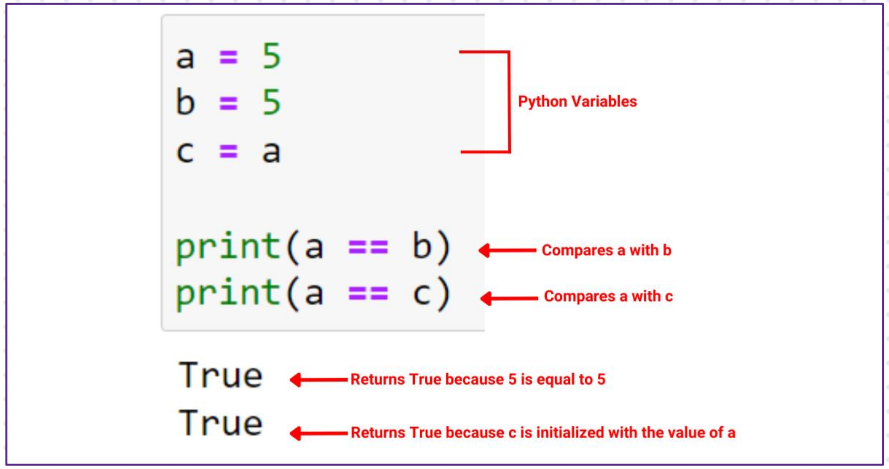
Conditionals and Recursion

CSC 1200 - Principles of Computing

Overview

- Booleans
- Modulo
- Logical Operators
- Truth Values
- Conditional Execution
- Compound Statements
- Keyboard Input
- Prompting the User
- Recursion

Boolean Expressions



```python
a = 5
b = 5
c = a

print(a == b)
print(a == c)
```

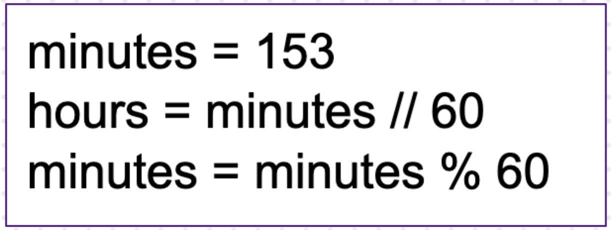
```
True
True
```

Python Variables
Compares a with b
Compares a with c
Returns True because 5 is equal to 5
Returns True because c is initialized with the value of a

- A **boolean expression** is an expression that is either True or False.
  - True and False are special values that belong to the Python type bool.
- **Relational operators** are used to make comparisons between values:
  - x < y evaluates to True if x is less than y and False otherwise
  - x <= y evaluates to True if x is less than or equal to y and False otherwise
  - x > y evaluates to True if x is greater than y and False otherwise
  - x >= y evaluates to True if x is greater than or equal to y and False otherwise
  - x == y evaluates to True if x is equal to y and False otherwise
  - x != y evaluates to True if x is NOT equal to y and False otherwise
- Note the difference between the **assignment operator =** and the **relational operator ==**. These are completely different operators, but easy to get mixed up!

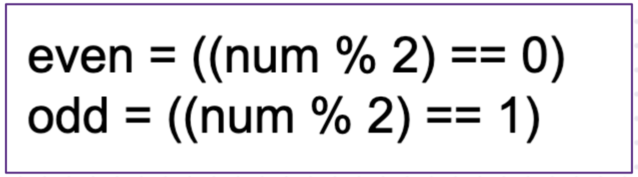
Usefulness of Modulo Operator

- We have already seen how the modulo operator is useful when determining the number of hours and minutes.

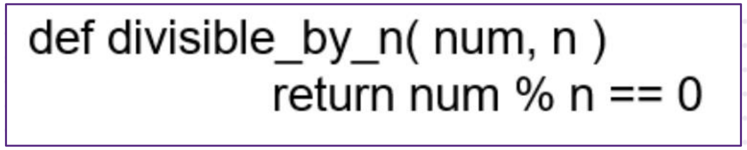


```
minutes = 153
hours = minutes // 60
minutes = minutes % 60
```

- Another common use of the modulo operator is to check if a number is even/odd or generally if it’s divisible by a certain number.

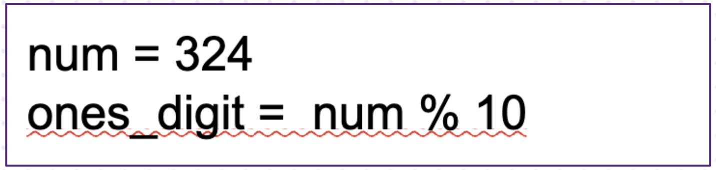


```
even = ((num % 2) == 0)
odd = ((num % 2) == 1)
```



```
def divisible_by_n( num, n )
            return num % n == 0
```

- The modulo operator is also commonly used to “extract” digits from a number.



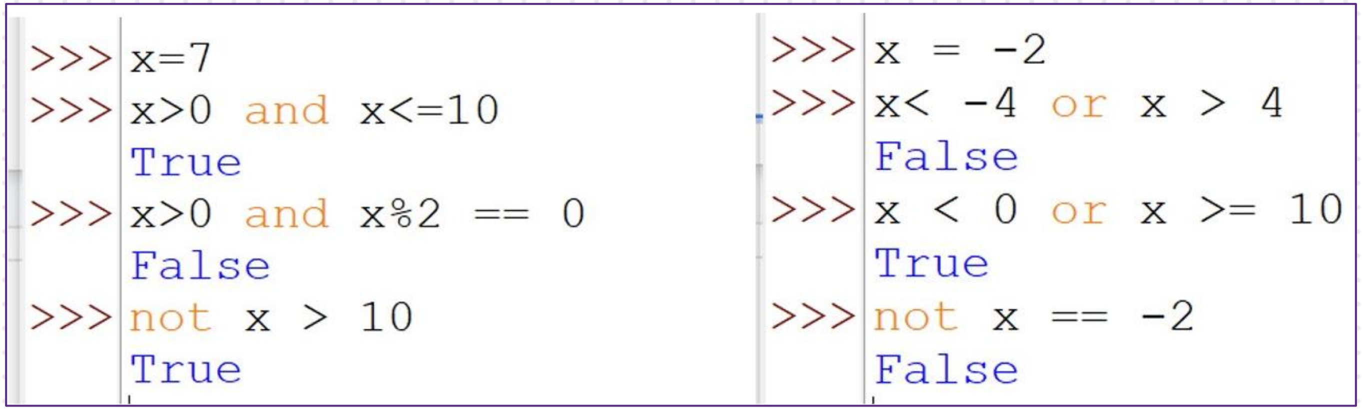
```
num = 324
ones_digit =  num % 10
```

Logical Operators

There are 3 **logical operators**:

- and → x and y is True only if BOTH x and y are True
- or → x or y is True if AT LEAST 1 of x and y is True
- not → gives the opposite truth value; not True is False and not False is True

Examples:



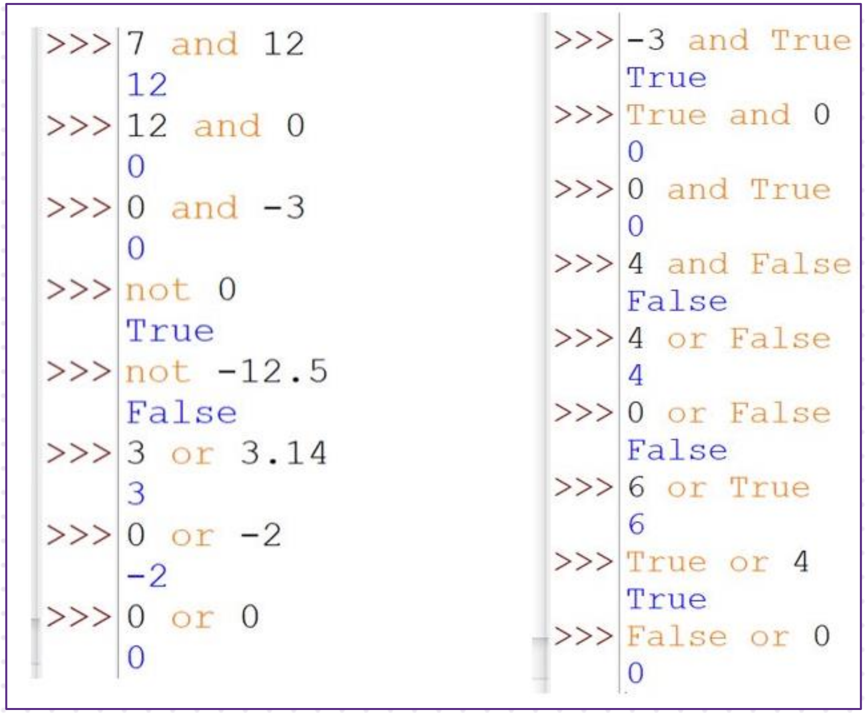
```
>>> x=7
>>> x>0 and x<=10
True
>>> x>0 and x%2 == 0
False
>>> not x > 10
True
```

```
>>> x = -2
>>> x< -4 or x > 4
False
>>> x < 0 or x >= 10
True
>>> not x == -2
False
```

Numbers Used as Truth Values

- Python treats 0 as False and all nonzero values as True

Examples: (these make sense if you “short circuit” the evaluations)



```
>>> 7 and 12
12
>>> 12 and 0
0
>>> 0 and -3
0
>>> not 0
True
>>> not -12.5
False
>>> 3 or 3.14
3
>>> 0 or -2
-2
>>> 0 or 0
0
```

```
>>> -3 and True
True
>>> True and 0
0
>>> 0 and True
0
>>> 4 and False
False
>>> 4 or False
4
>>> 0 or False
False
>>> 6 or True
6
>>> True or 4
True
>>> False or 0
0
```

```
True   and x    is x
False and x    is False

True  or x       is True
False or x      is x
```

Conditional Execution

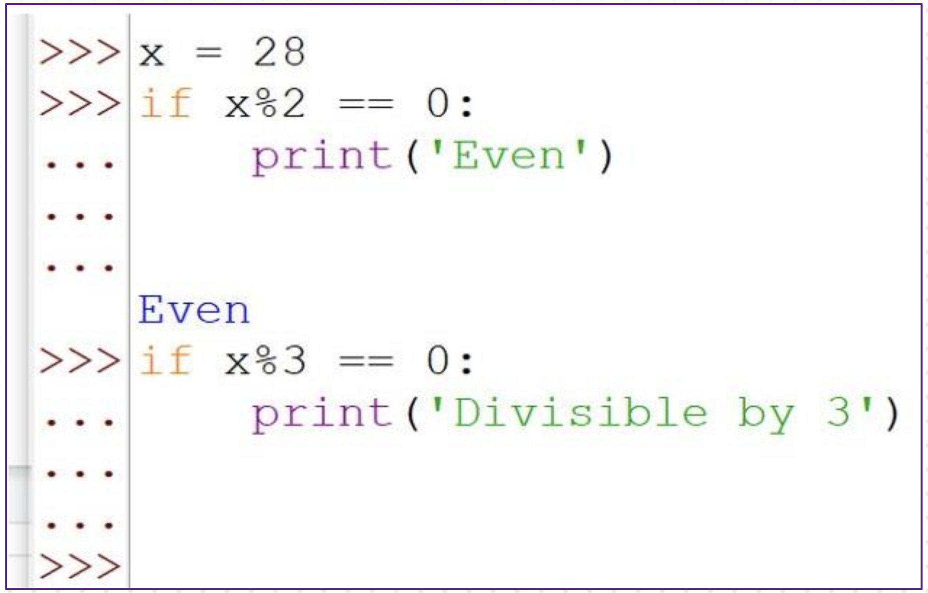
- To program anything significant, we need to ability to check for a condition and change the behavior of the program accordingly.
- **Conditional Statements** allow us to do this.
- General form of a conditional statement

  if condition :

  <tab> statement
- statement is ONLY done if the

  condition is True

- Example —----------------------------------->



```
>>> x = 28
>>> if x%2 == 0:
...     print('Even')
...
...
    Even
>>> if x%3 == 0:
...     print('Divisible by 3')
...
...
>>>
```

Compound Statements

- Notice that if statements have the same structure as function definitions:

```
def funct_name( param ):
            statement
            …
            statement
```

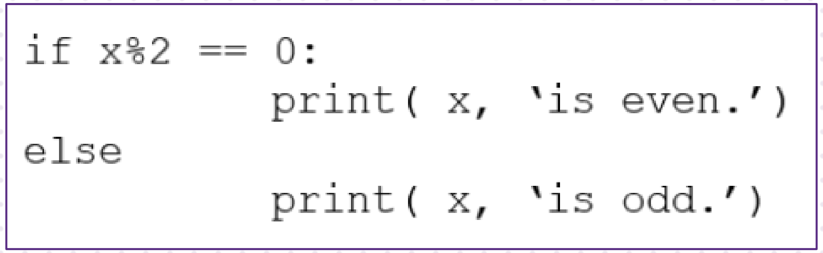
```
if condition:
            statement
            …
            statement
```

```
Header:
<tab>statement
…
<tab>statement
```

- Statements like this are called **compound statements**.
- There is no limit to the number of statements that can appear in the body, but there must be at least one.

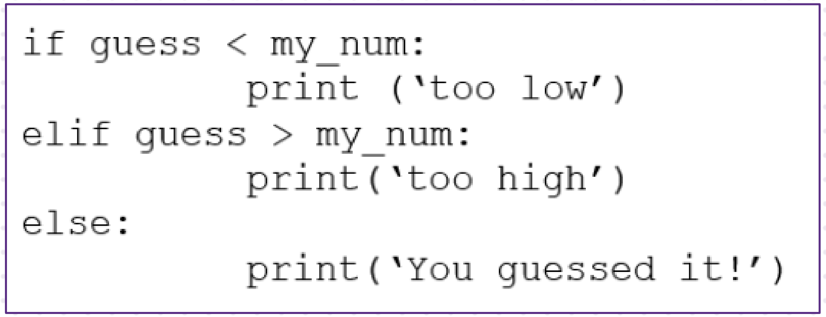
Alternative Execution and Chained Conditionals

- **Alternative execution:** there a 2 possibilities, and the condition determines which gets executed.



```
if x%2 == 0:
          print( x, ‘is even.’)
else
          print( x, ‘is odd.’)
```

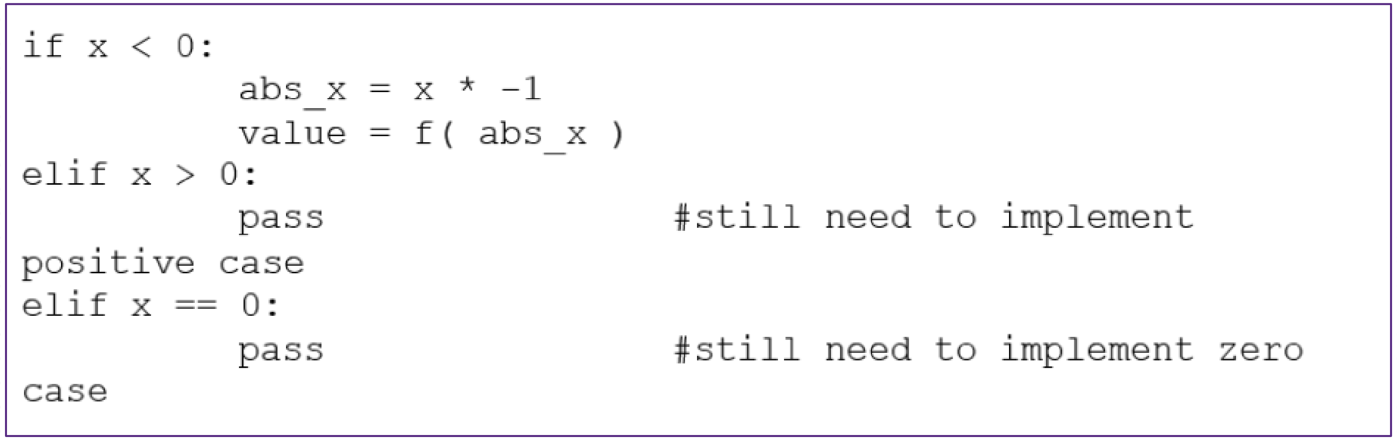
- **Chained conditionals:** there are more than 2 possibilities, and we need more than one conditional to determine the path taken



```
if guess < my_num:
          print (‘too low’)
elif guess > my_num:
          print(‘too high’)
else:
          print(‘You guessed it!’)
```

More on Conditional Statements

- You can use the pass statement, which does nothing, for an empty body or for a placeholder until you implement the body.

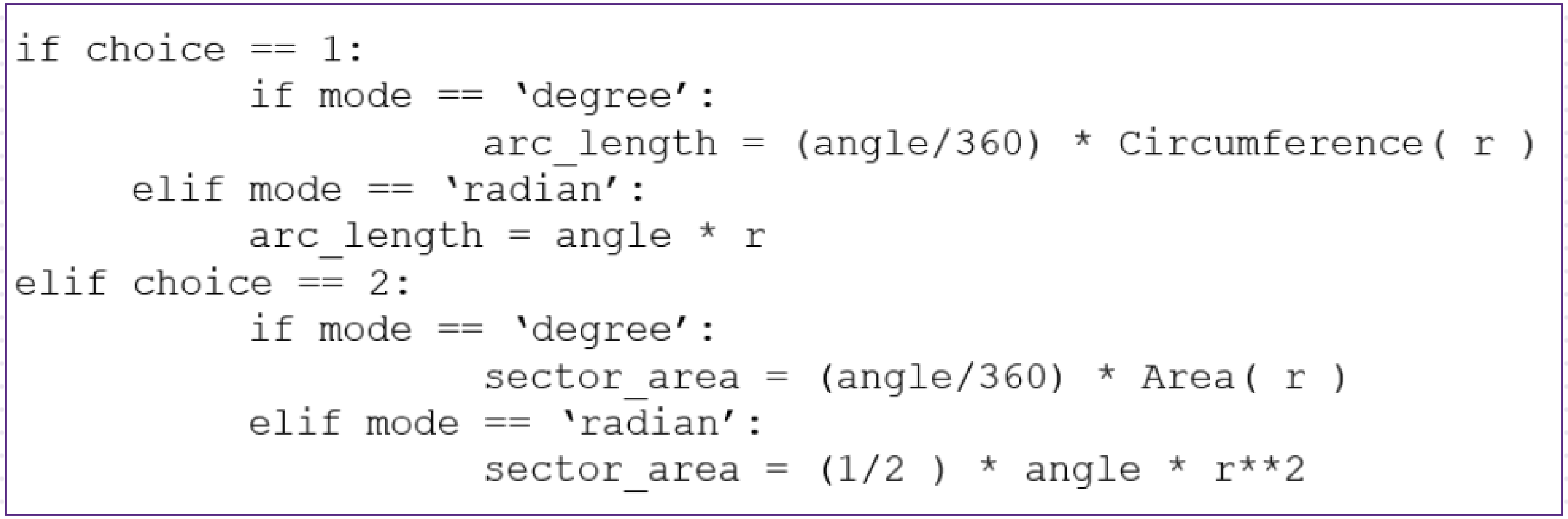


```
if x < 0:
          abs_x = x * -1
          value = f( abs_x )
elif x > 0:
          pass                #still need to implement
positive case
elif x == 0:
          pass                #still need to implement zero
case
```

- The above example shows that there does NOT have to be an else clause.

Nested Conditionals

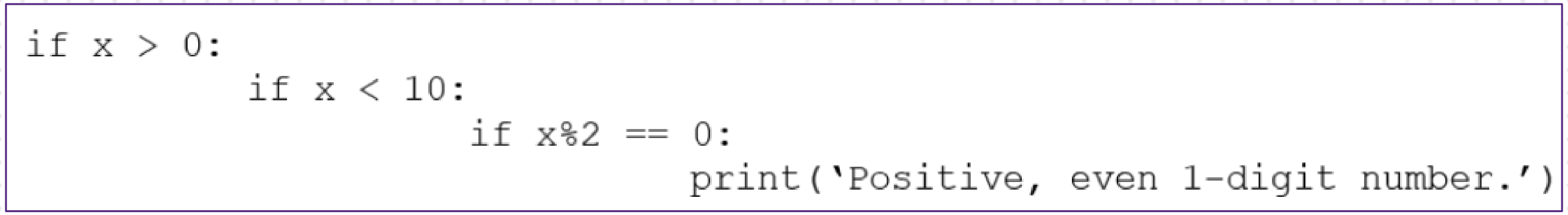
- A statement in the body of the if can be another conditional statement.



```
if choice == 1:
          if mode == ‘degree’:
                    arc_length = (angle/360) * Circumference( r )
     elif mode == ‘radian’:
          arc_length = angle * r
elif choice == 2:
          if mode == ‘degree’:
                    sector_area = (angle/360) * Area( r )
          elif mode == ‘radian’:
                    sector_area = (1/2 ) * angle * r**2
```

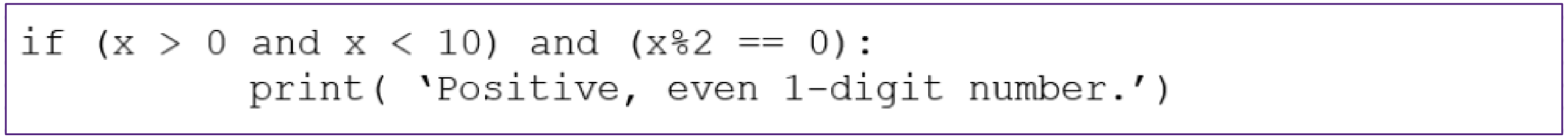
Nested Conditionals (Continued)

- Nested conditionals can be hard to follow, so use sparingly.



```
if x > 0:
        if x < 10:
                if x%2 == 0:
                        print(‘Positive, even 1-digit number.’)
```

is equivalent to…

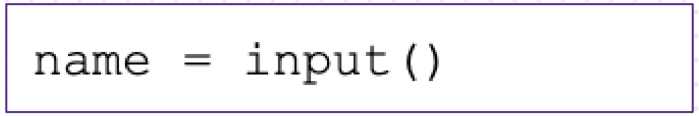


```
if (x > 0 and x < 10) and (x%2 == 0):
        print( ‘Positive, even 1-digit number.’)
```

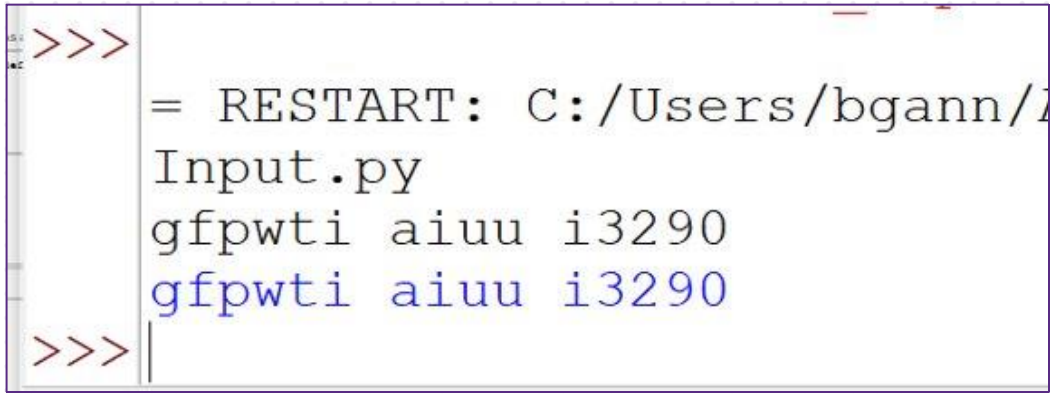
Keyboard Input

- So far, our programs have not had a way to interact with the user to get input.
- To get keyboard input in Python 3, we use the built-in function input.
- Note that the book describes keyboard input for Python 2 which uses raw_input. This function has been replaced in Python 3, so don't try to use raw_input.
- input will return a string containing whatever the user typed on the keyboard before pressing enter

Example:



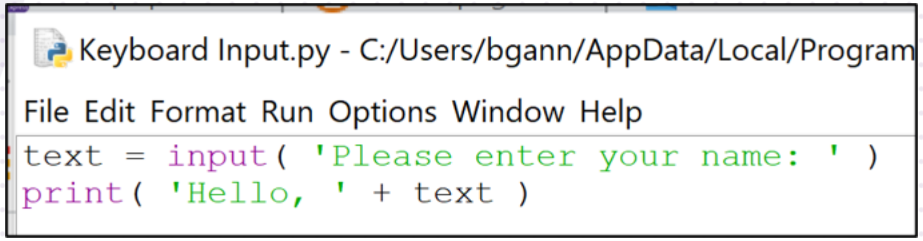
```
name = input()
```



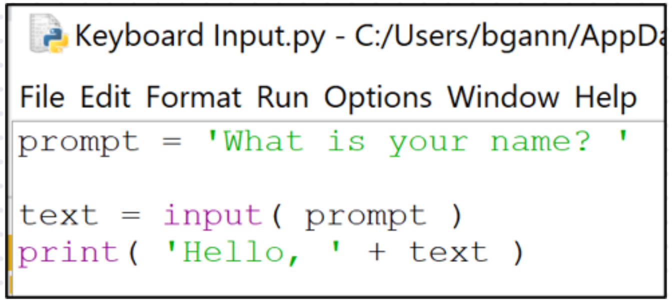
```
>>>
    = RESTART: C:/Users/bgann/
    Input.py
    gfpwti aiuu i3290
    gfpwti aiuu i3290
>>>
```

Prompting the User

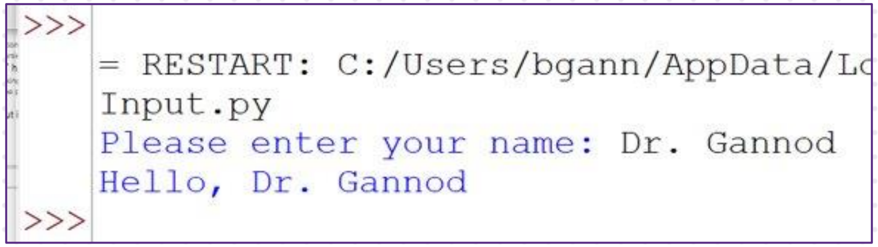
- In the previous example, the program just sits there waiting for the user to type something. What if the user doesn't know he is supposed to type something?
- As it is, the program is not **user-friendly**.
- Whenever the program is expecting input from the user, the program should prompt the user telling her what is expected.


Keyboard Input.py - C:/Users/bgann/AppData/Local/Program
File Edit Format Run Options Window Help

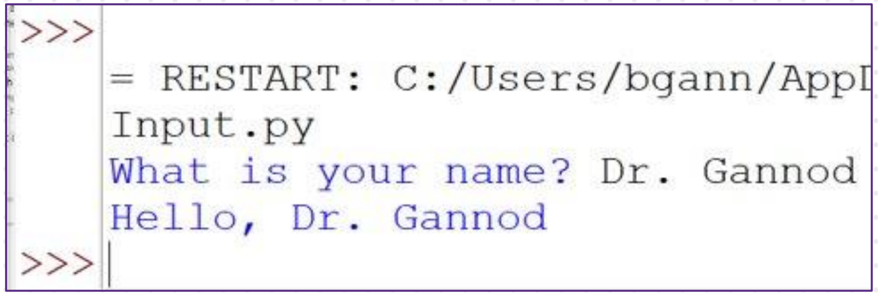
```
text = input( 'Please enter your name: ' )
print( 'Hello, ' + text )
```


Keyboard Input.py - C:/Users/bgann/AppDa
File Edit Format Run Options Window Help

```
prompt = 'What is your name? '

text = input( prompt )
print( 'Hello, ' + text )
```



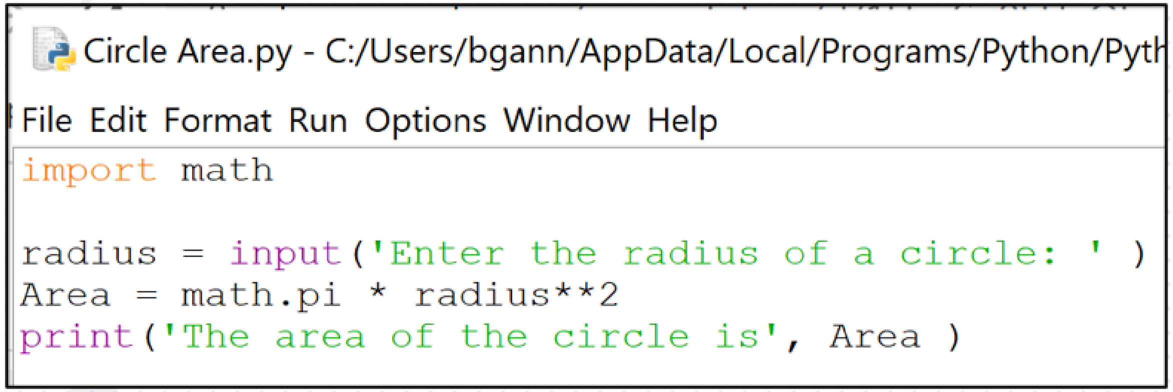
```
>>>
    = RESTART: C:/Users/bgann/AppData/Lo
    Input.py
    Please enter your name: Dr. Gannod
    Hello, Dr. Gannod
>>>
```



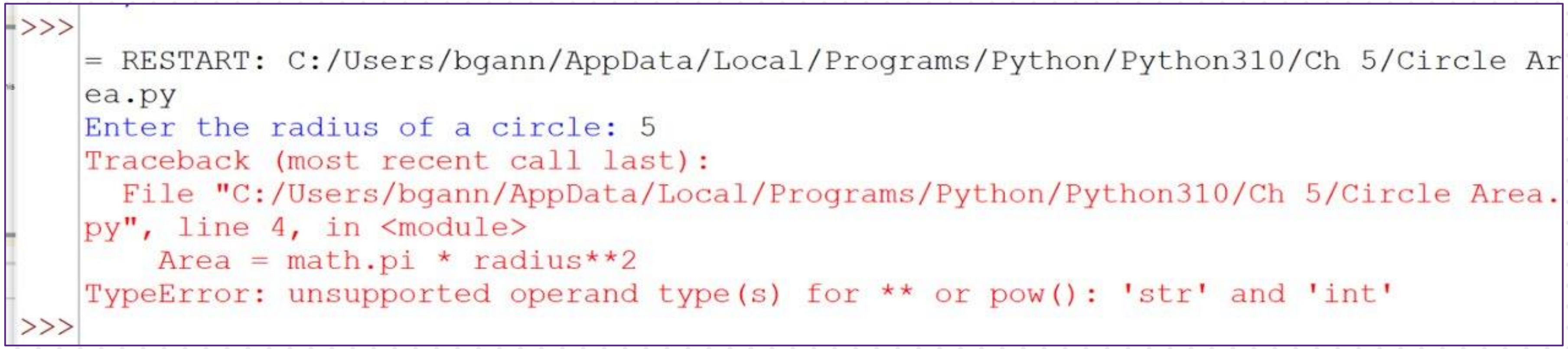
```
>>>
    = RESTART: C:/Users/bgann/AppD
    Input.py
    What is your name? Dr. Gannod
    Hello, Dr. Gannod
>>>
```

What If I Don’t Want A String?

Note: input ALWAYS returns a string. This can cause problems if that’s not what you want.


Circle Area.py - C:/Users/bgann/AppData/Local/Programs/Python/Pyth
File Edit Format Run Options Window Help

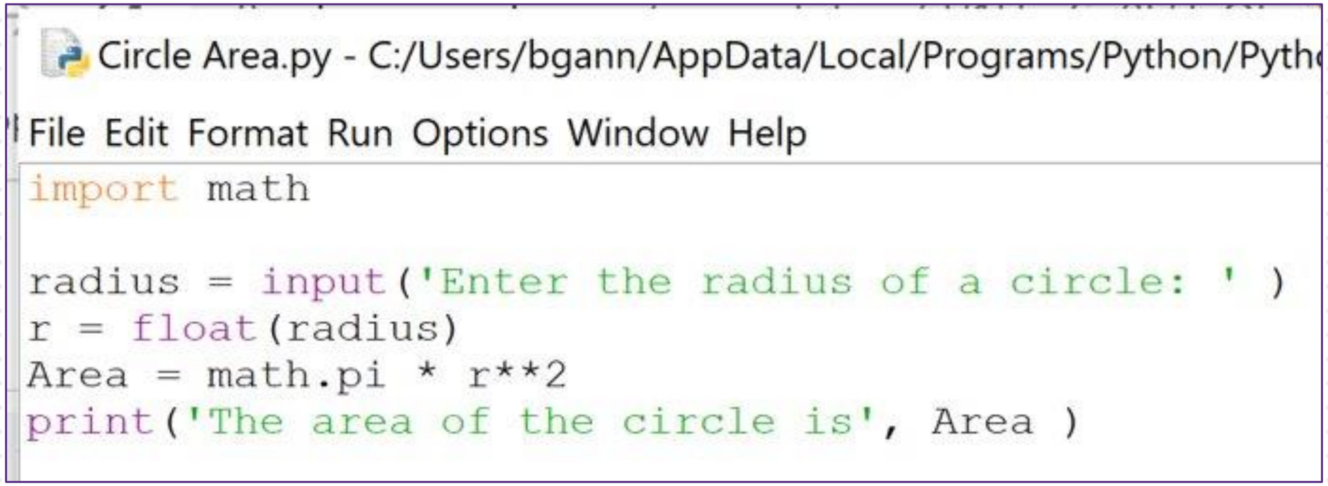
```
import math

radius = input('Enter the radius of a circle: ' )
Area = math.pi * radius**2
print('The area of the circle is', Area )
```



```
>>>
= RESTART: C:/Users/bgann/AppData/Local/Programs/Python/Python310/Ch 5/Circle Ar
ea.py
Enter the radius of a circle: 5
Traceback (most recent call last):
  File "C:/Users/bgann/AppData/Local/Programs/Python/Python310/Ch 5/Circle Area.
py", line 4, in <module>
    Area = math.pi * radius**2
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
>>>
```

Casting Input to Correct Type

- You can use the built-in Python functions `int()` or `float()` to **cast** the string to the correct type.



Circle Area.py - C:/Users/bgann/AppData/Local/Programs/Python/Pyth
File Edit Format Run Options Window Help

```
import math

radius = input('Enter the radius of a circle: ' )
r = float(radius)
Area = math.pi * r**2
print('The area of the circle is', Area )
```

Recursion

- We have seen that functions can call other functions. Functions can also call themselves! This is called **recursion**.
- Many problems can be broken down into a simple action for part of the problem and a smaller version of the same problem.
- Example: Countdown from n to 0
  - Say 'n' ← Simple action for part
  - Then Countdown from n-1 to 0 ← Smaller version of the same problem
- A recursive solution to a problem MUST have two important components:
  - A base case – this tells us when to stop breaking the problem down
  - A recursive rule – this tells us how to break the problem down into the simple action and smaller version of the problem.
- The base case is very important. Without it, we will NEVER stop (well…at least not until the program crashes!)

Recursive Example

Countdown from n to blastoff

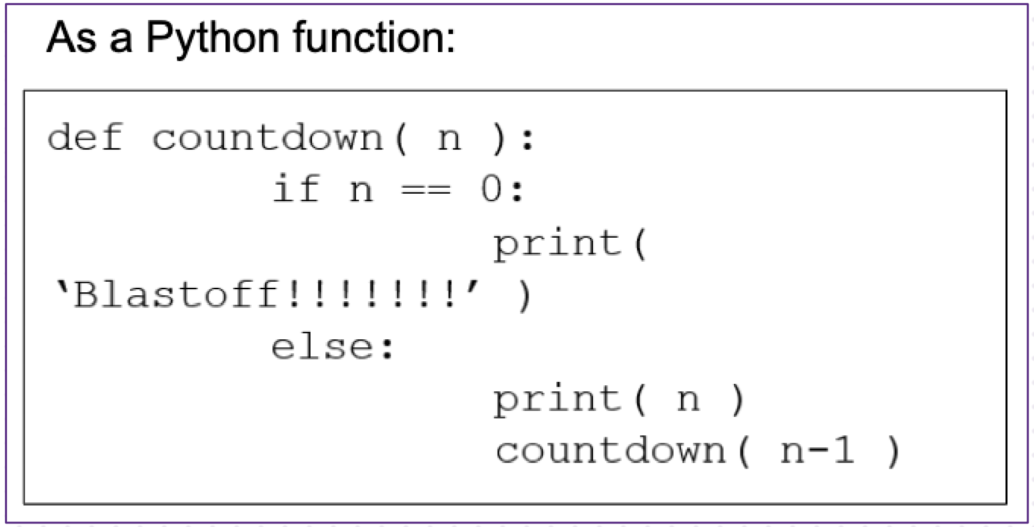
When do we stop (base case)?

When n = 0, blastoff

How do we break the problem down?

Say ‘n’

Countdown from n-1 to blastoff



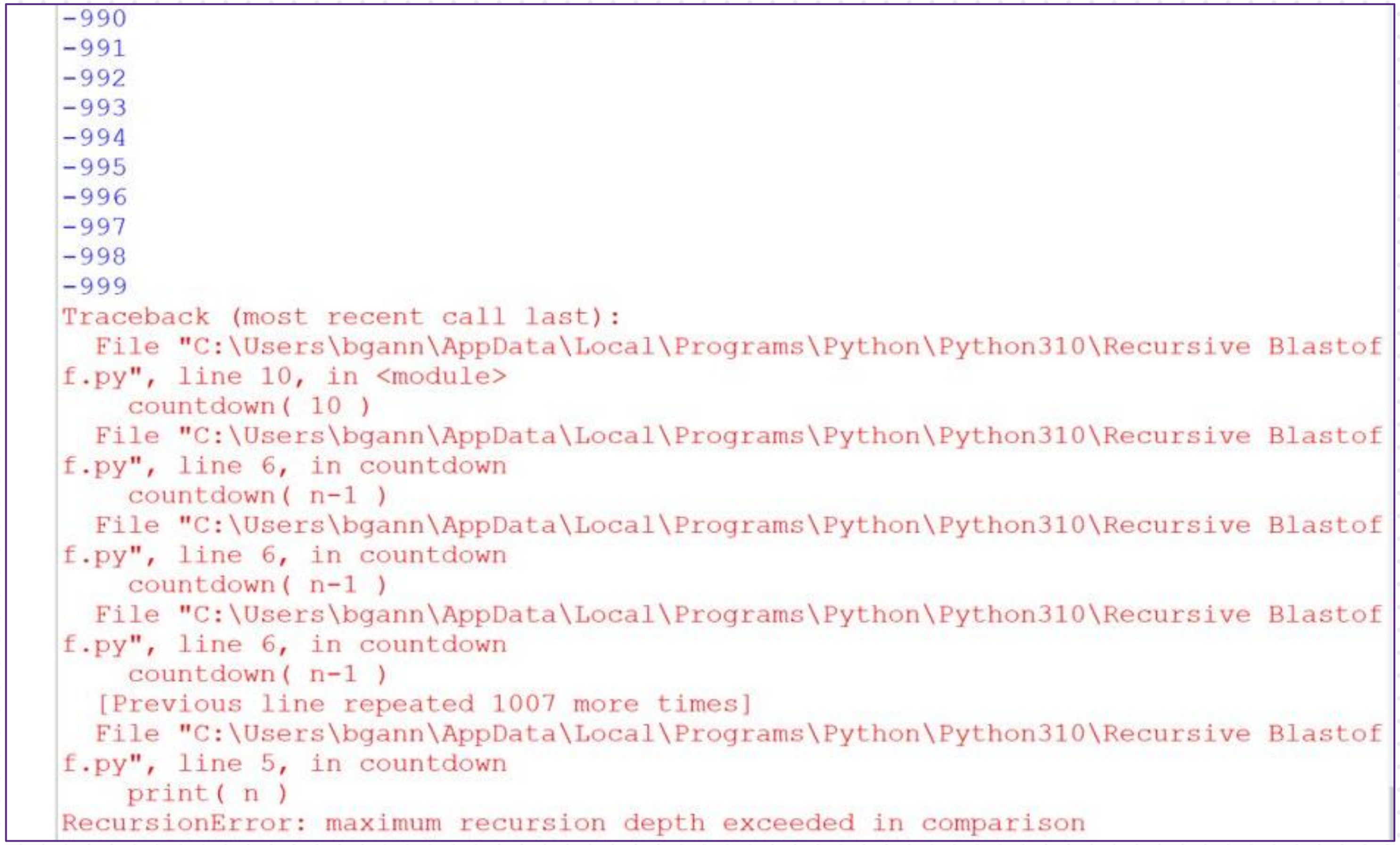
As a Python function:

```
def countdown( n ):
         if n == 0:
                  print(
‘Blastoff!!!!!!!’ )
         else:
                  print( n )
                  countdown( n-1 )
```

Infinite Recursion

What happens if we don't include a base case? Infinite recursion

```
def countdown( n ):
        print( n )
        countdown(n-1)
```



```
-990
-991
-992
-993
-994
-995
-996
-997
-998
-999
Traceback (most recent call last):
  File "C:\Users\bgann\AppData\Local\Programs\Python\Python310\Recursive Blastof
f.py", line 10, in <module>
    countdown( 10 )
  File "C:\Users\bgann\AppData\Local\Programs\Python\Python310\Recursive Blastof
f.py", line 6, in countdown
    countdown( n-1 )
  File "C:\Users\bgann\AppData\Local\Programs\Python\Python310\Recursive Blastof
f.py", line 6, in countdown
    countdown( n-1 )
  File "C:\Users\bgann\AppData\Local\Programs\Python\Python310\Recursive Blastof
f.py", line 6, in countdown
    countdown( n-1 )
  [Previous line repeated 1007 more times]
  File "C:\Users\bgann\AppData\Local\Programs\Python\Python310\Recursive Blastof
f.py", line 5, in countdown
    print( n )
RecursionError: maximum recursion depth exceeded in comparison
```