Sentinel Loops

- What if we want to average a set of positive and negative numbers?
- Valid input would be converted into numeric form
- We could use the *empty string* ("" or '') as the sentinel

Sentinel Loops

```
# average4.py

def main():
    sum = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit)>>")
    while xStr != "":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit)>>")
    print("The average of the numbers is", sum / count)

main()
```

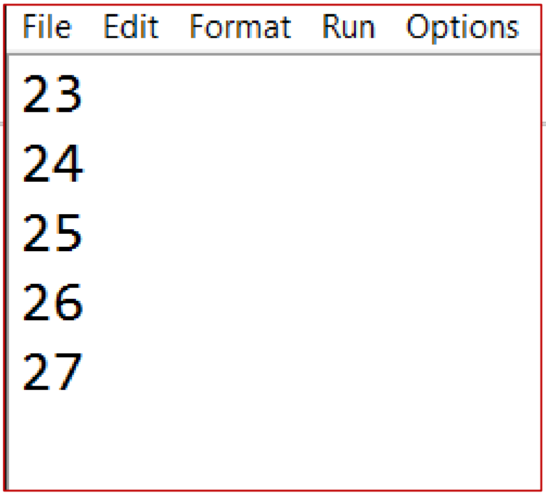
Sentinel Loops

```
Enter a number (<Enter> to quit): 34
Enter a number (<Enter> to quit): 23
Enter a number (<Enter> to quit): 0
Enter a number (<Enter> to quit): -25
Enter a number (<Enter> to quit): -34.4
Enter a number (<Enter> to quit): 22.7
Enter a number (<Enter> to quit):
The average of the numbers is 3.38333333333
```

File Loops

- Above programs are all interactive
- What happens if you made a typo on number 85 out of 100?

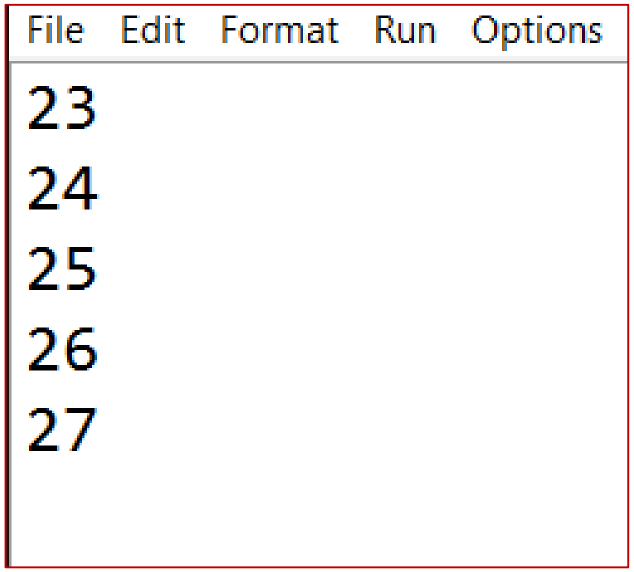
File Loops



```
23
24
25
26
27
```

```
# average5.py

def main():
    fileName = input("What file are the numbers in?")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    for line in infile:
        sum = sum + eval(line)
        count = count + 1
    print("The average of the numbers is", sum/count)

main()
```

Sentinel Loops



```
23
24
25
26
27
```

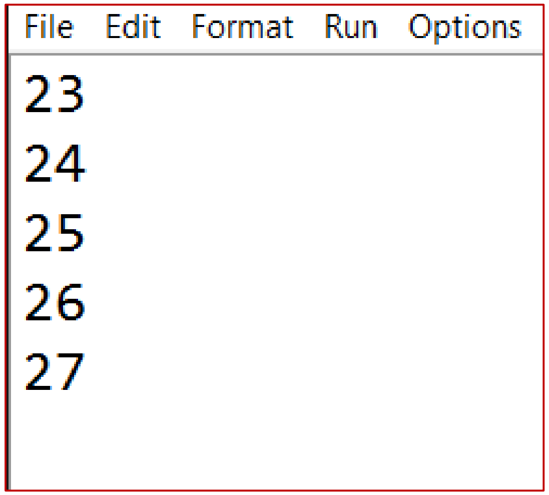
```
What file are the numbers in?nums.txt
The average of the numbers is 25.0
```

File Loops

- We could use `readline` in a sentinel loop to get the next line of the file
- At the end of the file, `readline` returns an empty string, “”

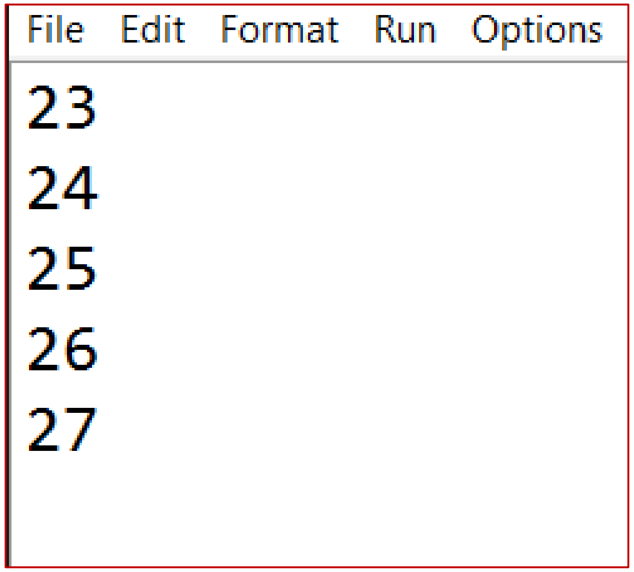
File Loops

```
# average6.py

def main():
    fileName = input("What file are the numbers in?")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        sum = sum + eval(line)
        count = count + 1
        line = infile.readline()
    print("The average of the numbers is", sum/count)

main()
```



```
23
24
25
26
27
```

Sentinel Loops

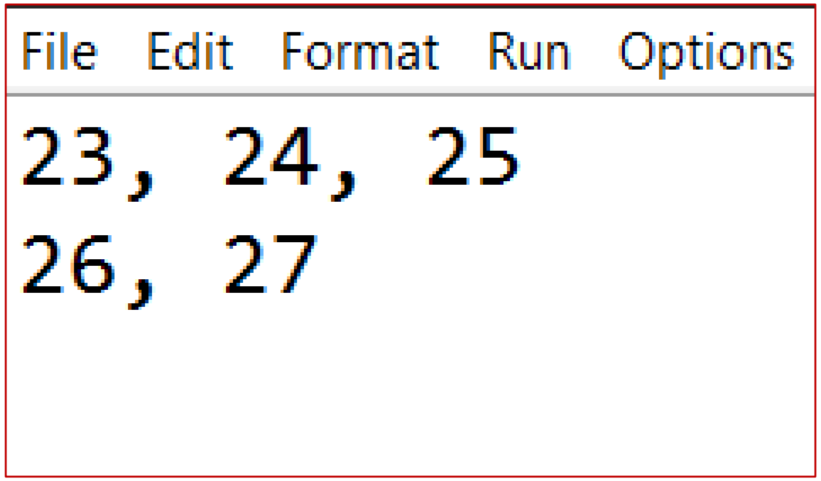


```
23
24
25
26
27
```

```
What file are the numbers in?nums.txt
The average of the numbers is 25.0
```

Nested Loops

- Suppose there are multiple numbers in a line (separated by commas), rather than one number per line



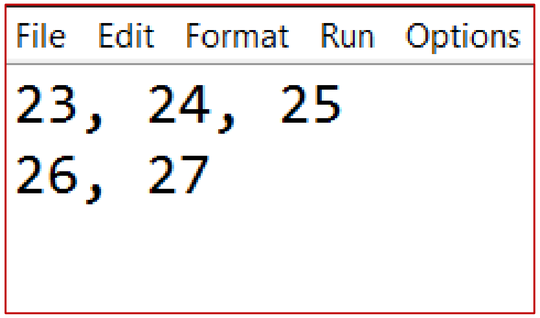
```
23, 24, 25
26, 27
```

Nested Loops

- Split the string into substrings, each of which represents a number
- Loop through the substrings, convert each to a number, and add it to `sum`
- Update `count`

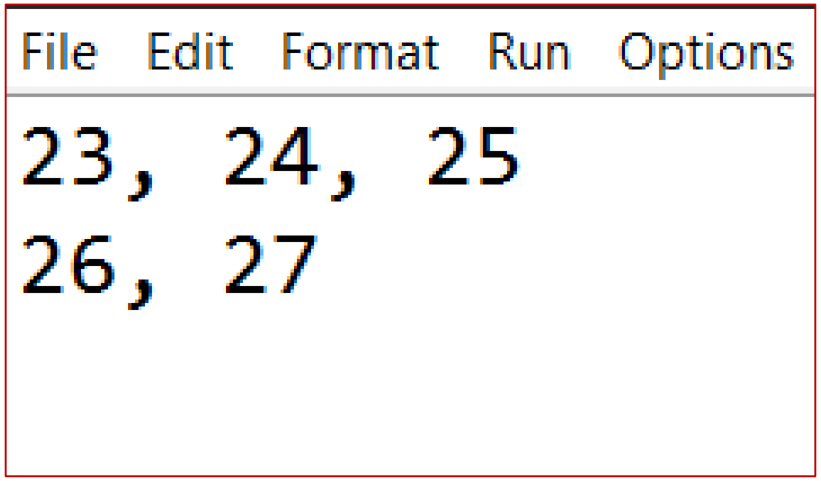
Nested Loops

```
# average7.py

def main():
    fileName = input("What file are the numbers in?")
    infile = open(fileName,'r')
    sum = 0.0
    count = 0
    line = infile.readline()
    while line != "":
        # update sum and count for values in line
        for xStr in line.split(","):
            sum = sum + eval(xStr)
            count = count + 1
        line = infile.readline()
    print("The average of the numbers is", sum/count)

main()
```



```
23, 24, 25
26, 27
```

Sentinel Loops



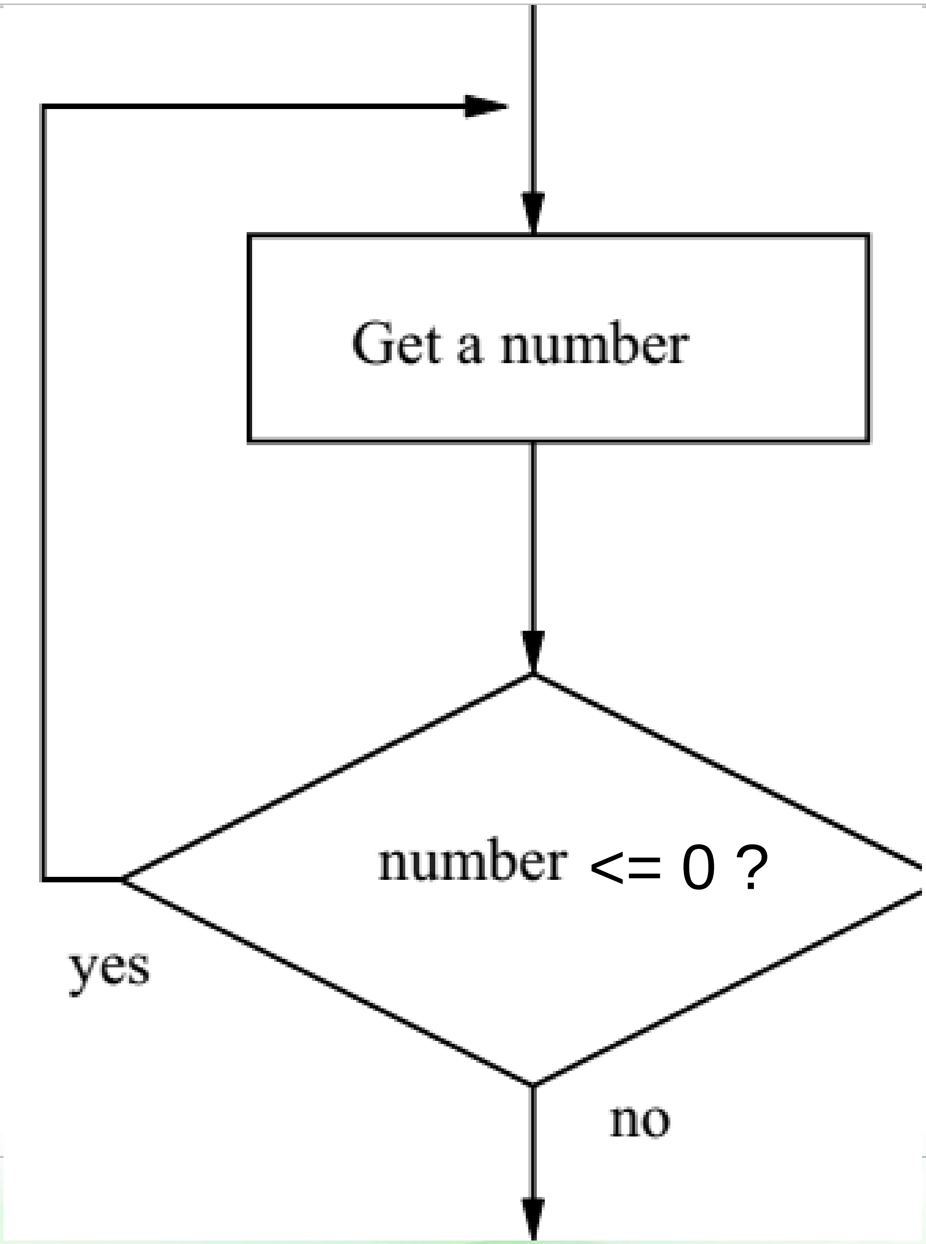
```
23, 24, 25
26, 27
```

```
What file are the numbers in?nums2.txt
The average of the numbers is 25.0
```

Input Validation

- Write a program that takes a positive number from the user
- If the user input is not valid (**zero or negative**), asks for another value, until a valid value has been entered

Input Validation



BINGHAMTON UNIVERSITY STATE UNIVERSITY OF NEW YORK

Input Validation

```python
# post_test.py
def main():
    number = -1
    while number <= 0:
        number = eval(input("Enter a positive number:"))
    print("The positive number is", number)

main()
```

Input Validation

```
Enter a positive number:0
Enter a positive number:-3
Enter a positive number:2
The positive number is 2
```

Input Validation

- Executing Python `break` statement causes the program to immediately exit the enclosing loop
- For nested loops, if `break` is in the innermost loop, it will break out from only the innermost loop

Input Validation

```
# post_test2.py
def main():
    while True:
        number = eval(input("Enter a positive number:"))
        if number > 0:
            break    # Exit loop if number is valid

    print("The positive number is", number)

main()
```

Input Validation

```
Enter a positive number:0
Enter a positive number:-3
Enter a positive number:2
The positive number is 2
```

Input Validation

- Give a warning when the input was invalid

```python
# post_test4.py
def main():
    while True:
        number = eval(input("Enter a positive number:"))
        if number > 0:
            break  # Loop exit
        print("The number was not positive!")

    print("The positive number is", number)

main()
```

Input Validation

```
Enter a positive number:0
The number was not positive!
Enter a positive number:-3
The number was not positive!
Enter a positive number:2
The positive number is 2
```

Use of Breaks

- The use of `break` is mostly a matter of style and taste
- Avoid using break often within loops
- The logic of loops is hard to follow when there are multiple exits

Computing with Booleans

- `if` and `while` both use Boolean expressions
- Boolean expressions evaluate to `True` or `False`

Computing with Booleans

- Simple Boolean expressions compare two values
  - `while x >= 0:`
  - `while y == 0:`
- What if you want to check whether both conditions hold at the same time?

Boolean Operators

- The `and` of two expressions is true only if both expressions are true
- Similar to math multiplication

P	***Q***	***P and Q***
True	True	True
True	False	False
False	True	False
False	False	False

Boolean Expressions

- The `or` of two expressions is true when either expression is true
- Similar to math addition

P	***Q***	***P or Q***
True	True	True
True	False	True
False	True	True
False	False	False

Boolean Operators

- The `not` operator computes the opposite of a Boolean expression
- `not` is a *unary* operator, meaning it operates on a single expression

P	***not P***
True	False
False	True

Boolean Operators

- We can put these operators together to make complex Boolean expressions
- The interpretation of the expressions relies on the *precedence rules* for the operators

Boolean Operators

```
>>> a = True
>>> a
True
>>> b = 3 >= 5
>>> b
False
>>> c = " " != ""
>>> c
True
```

Boolean Operators

- The order of precedence, from high to low, is `not`, `and`, `or`

```
>>> a, b, c = True, False, True
>>> a or not b and c
```

???

- **Use parentheses to prevent confusion**

Boolean Algebra

- Both `and` and `or` distribute:

```
a or (b and c) == (a or b) and (a or c)
a and (b or c) == (a and b) or (a and c)
```

- Double negatives cancel out:

```
not(not a) == a
```

- DeMorgan’s laws:

```
not(a or b) == (not a) and (not b)
not(a and b) == (not a) or (not b)
```

Boolean Algebra

- In a program that simulates a racquetball game, the game is over as soon as either Player A or Player B has scored 15 points

```
scoreA == 15 or scoreB == 15
```

Boolean Algebra

- The condition that a game is `not` over

```
not(scoreA == 15 or scoreB == 15)
```

Boolean Algebra

- After applying the DeMorgan’s law, we get the following equivalent expression

```
(not scoreA == 15) and (not scoreB == 15)
```

- Further simplified

```
(scoreA != 15) and (scoreB != 15)
```

Convert Built-in Data Types to Boolean

- Check if a user's input starts with 'y' or 'Y'

```
response[0] == "y" or response[0] == "Y"
```

Convert Built-in Data Types to Boolean

- You can’t take shortcuts:

```
response[0] == "y" or "Y"
```

Convert Built-in Data Types to Boolean

- For a number (`int` or `float`)
  - zero: `False`
  - anything else: `True`
- For a sequence (`string, list, tuple, dict, set`)
  - empty: `False`
  - non-empty: `True`

Convert Built-in Data Types to Boolean

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool(-2.1)
True
>>> bool("Hello")
True
>>> bool("")
False
>>> bool([1,2,3])
True
>>> bool(())
False
```

Convert Built-in Data Types to Boolean

- The following two are equivalent:

```
response[0] == "y" or "Y"
(response[0] == "y") or ("Y")
```

- Because “Y” is always `True`, the `or` operation is also always `True`

Convert Built-in Data Types to Boolean

- Boolean operators are *short-circuit* operators
- A True or False is returned as soon as the result is known

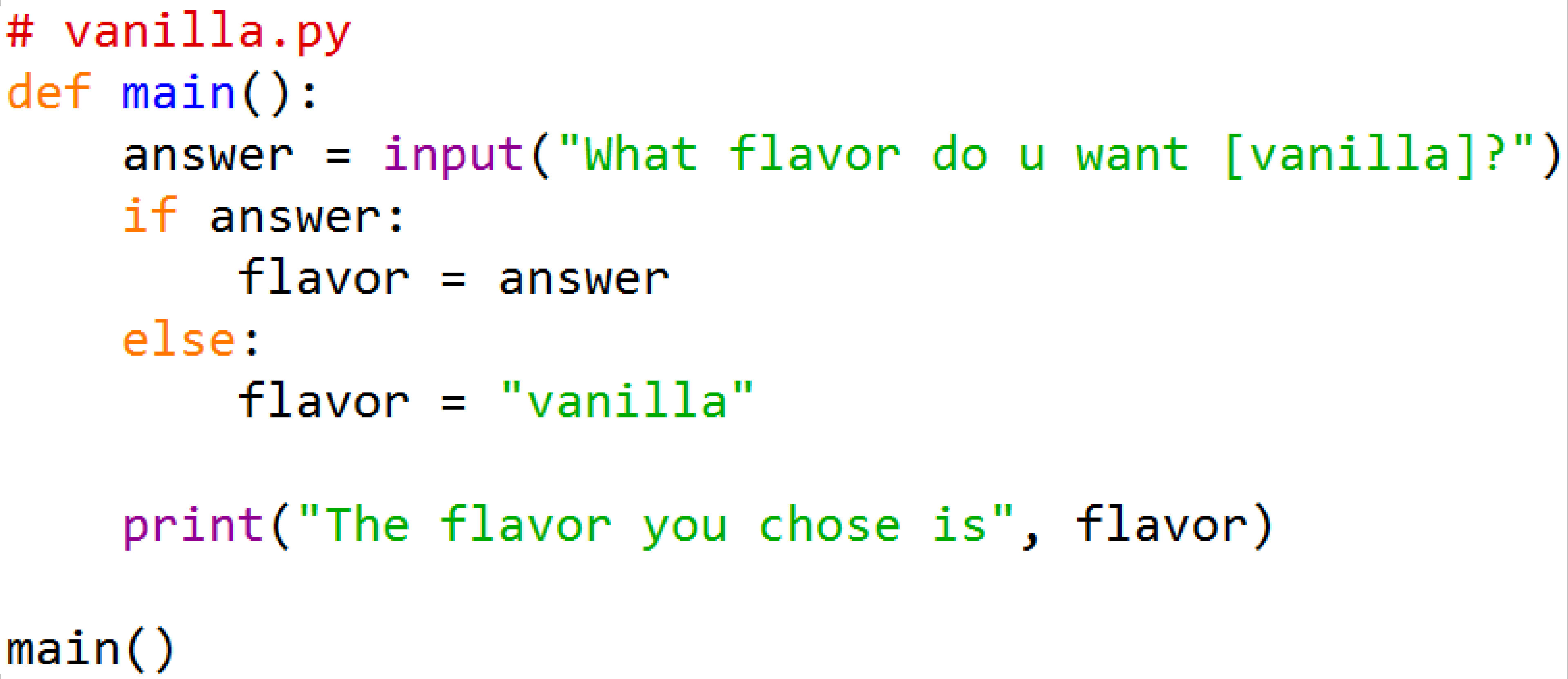
Convert Built-in Data Types to Boolean

Operator	Operational definition
x `and` *y*	If *x* is false, return `False`. Otherwise, return *y*.
x `or` *y*	If *x* is true, return `True`. Otherwise, return *y*.
`not` *x*	If *x* is false, return `True`. Otherwise, return `False`.

Convert Built-in Data Types to Boolean

- Write a program that request for information
- Offer a default value when the user simply pressing `<Enter>`

Convert Built-in Data Types to Boolean



```
# vanilla.py
def main():
    answer = input("What flavor do u want [vanilla]?")
    if answer:
        flavor = answer
    else:
        flavor = "vanilla"

    print("The flavor you chose is", flavor)

main()
```

Convert Built-in Data Types to Boolean

```
What flavor do u want [vanilla]?
The flavor you chose is vanilla

What flavor do u want [vanilla]?chocolate
The flavor you chose is chocolate
```

Boolean Expressions as Decisions

- A even more succinct program

```
# vanilla2.py
def main():
    answer = input("What flavor do u want [vanilla]?")
    flavor = answer or "vanilla"
    print("The flavor you chose is", flavor)

main()
```

- When you code is this tricky, make sure it is well documented

Convert Built-in Data Types to Boolean

```
What flavor do u want [vanilla]?
The flavor you chose is vanilla

What flavor do u want [vanilla]?chocolate
The flavor you chose is chocolate
```

Using Python to Code by Voice