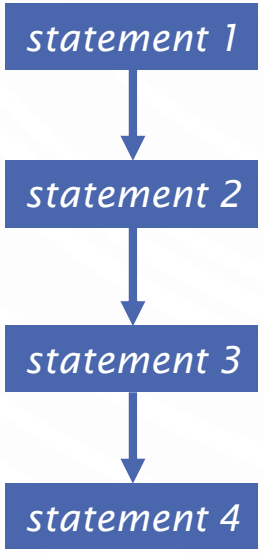
CHAPTER 5
LOOPS

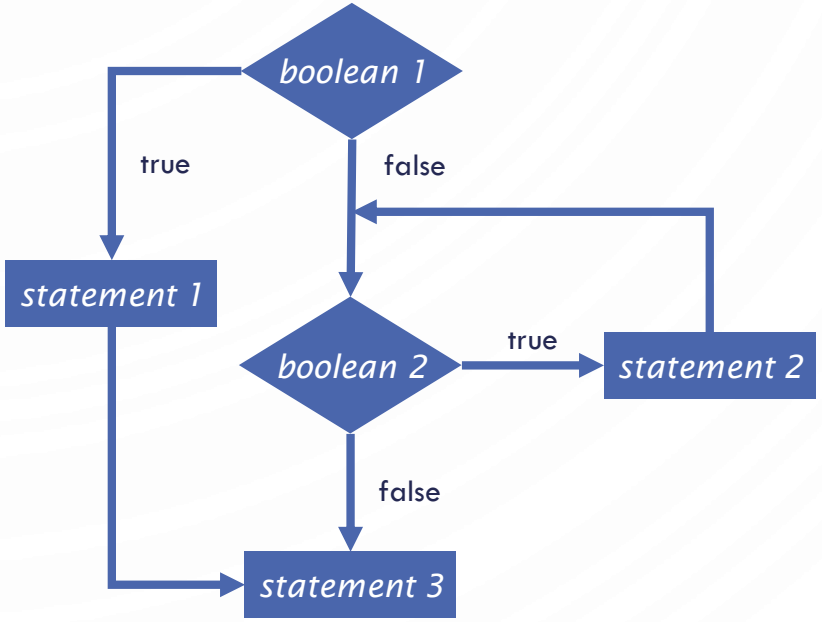
ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO PROGRAMMING USING PYTHON, LIANG (PEARSON 2013)

CONTROL FLOW

- Control flow.
  - Sequence of statements that are actually executed in a program.
  - Conditionals and loops: enable us to choreograph control flow.



straight-line control flow



control flow with conditionals and loops

MOTIVATIONS

- Suppose that you need to print a string (e.g., "Welcome to Python!") a hundred times. It would be tedious to have to write the following statement a hundred times:

```
print("Welcome to Python!")
```

  So, how do you solve this problem?

- How about altering our guessing game program to allow 20 tries?

OPENING PROBLEM

```
print("Welcome to Python!")
print("Welcome to Python!")
print("Welcome to Python!")
print("Welcome to Python!")
print("Welcome to Python!")
print("Welcome to Python!")
...
print("Welcome to Python!")
print("Welcome to Python!")
```

100 times

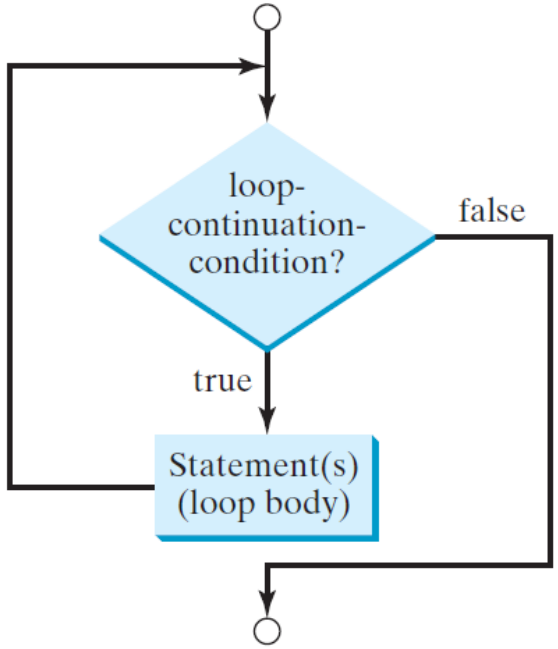
THE WHILE LOOP

INTRODUCING WHILE LOOPS

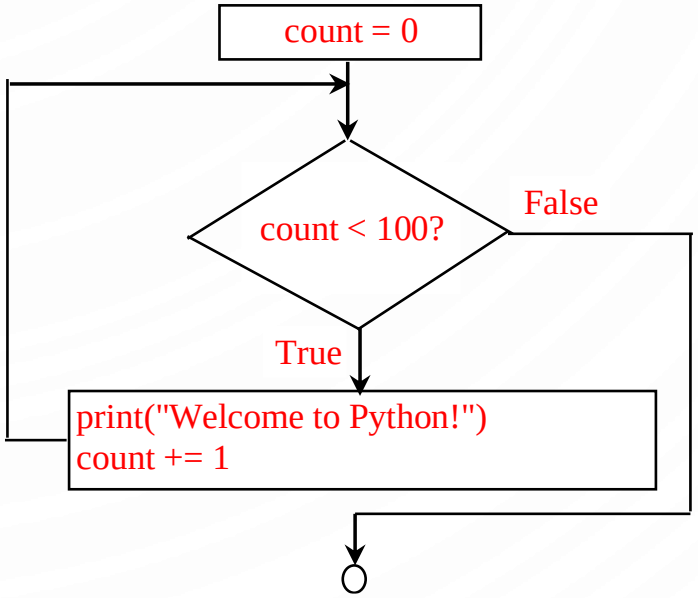
```
1. count = 0
2. while count < 100:
3.     print("Welcome to Python")
4.     count += 1
```

WHILE LOOP FLOW CHART

```
1.  while loop-continuation-condition:
2.      # loop-body
3.      Statement(s)
```




```
1.  count = 0
2.  while count < 100:
3.      print("Welcome to Python")
4.      count += 1
```



TRACING WHILE LOOPS

```
1.  count = 0
2.  while count < 2:
3.      print("Welcome to Python")
4.      count += 1
```

Memory

Output

TRACING WHILE LOOPS

```
1. count = 0
2. while count < 2:
3.     print("Welcome to Python")
4.     count += 1
```

Initialize Count

Memory

count: 0

Output

TRACING WHILE LOOPS

```
1.  count = 0
2.  while count < 2:
3.      print("Welcome to Python")
4.      count += 1
```

Count < 2 is true

Memory

count: 0

Output

TRACING WHILE LOOPS

```
1.  count = 0
2.  while count < 2:
3.      print("Welcome to Python")
4.      count += 1
```

Output

Memory
count: 0

Output
Welcome to Python

TRACING WHILE LOOPS

```
1.  count = 0
2.  while count < 2:
3.      print("Welcome to Python")
4.      count += 1
```

Increment count

Memory

count: ~~0~~ 1

Output

Welcome to Python

TRACING WHILE LOOPS

```
1.  count = 0
2.  while count < 2:
3.      print("Welcome to Python")
4.      count += 1
```

Count < 2 is true

Memory

count: ~~0~~ 1

Output

Welcome to Python

TRACING WHILE LOOPS

```
1.  count = 0
2.  while count < 2:
3.      print("Welcome to Python")
4.      count += 1
```

Output

Memory

count: ~~0~~ 1

Output

Welcome to Python

Welcome to Python

TRACING WHILE LOOPS

```
1.  count = 0
2.  while count < 2:
3.      print("Welcome to Python")
4.      count += 1
```

Increment count

Memory

count: ~~0~~ ~~1~~ 2

Output

Welcome to Python
Welcome to Python

TRACING WHILE LOOPS

```
1.  count = 0
2.  while count < 2:
3.      print("Welcome to Python")
4.      count += 1
```

Count < 2 is false

Memory

count: ~~0~~ ~~1~~ 2

Output

Welcome to Python
Welcome to Python

TRACING WHILE LOOPS

```
1.  count = 0
2.  while count < 2:
3.      print("Welcome to Python")
4.      count += 1
```

Memory

count: ~~0~~ ~~1~~ 2

Output

Welcome to Python

Welcome to Python

EXAMPLES – WITH A PARTNER

- What are the values of n and m after this program:

```
n = 1234567
m = 0
while n != 0:
    m = (10*m) + (n % 10)
    n //= 10
```

- Show the trace of the program at each step

QUESTION

- What is wrong with the following code?
- What happens?
- Fix it and explain what the code outputs

```
1.  i, N = 0, 10000
2.  while i <= N:
3.      print(i)
4.  i = i + 5
```

ACTIVITY

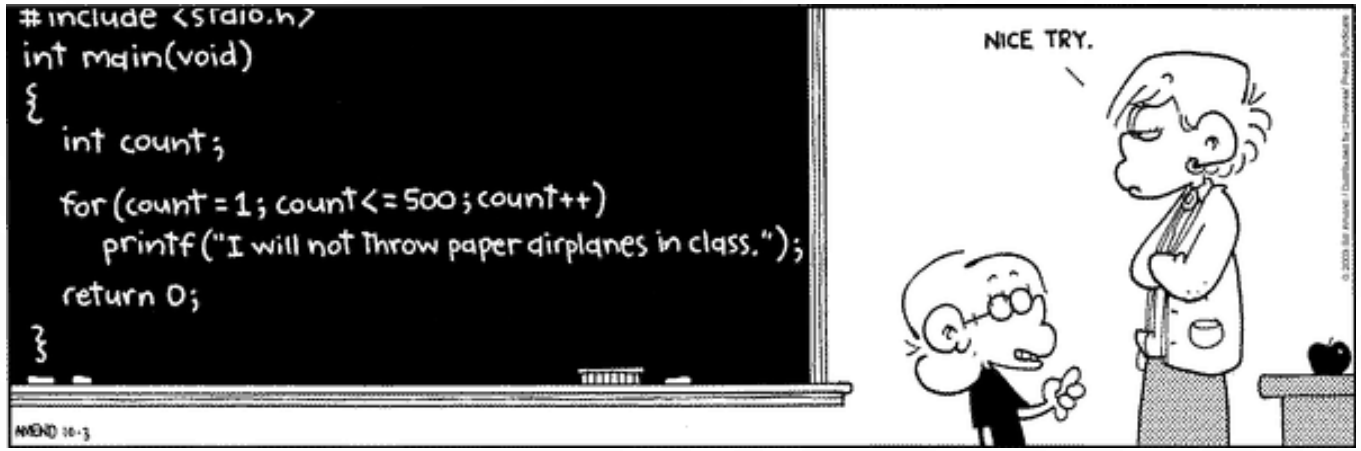
- Write an algorithm to compute the number of digits an integer has.
  - Example: input – 34567 output – 5
- Bonus: modify your algorithm to compute the number of “digits” that the number would have if converted to another base, e.g., binary, octal, or hexadecimal

CAUTION

- Don't use floating-point values for equality checking in a loop control. Since floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results. Consider the following code for computing 1 + 0.9 + 0.8 + ... + 0.1:

```
1. item, sum = 1, 0
2. while item != 0: # No guarantee item will be 0
3.     sum += item
4.     item -= 0.1
5. print(sum)
```

THE FOR LOOP

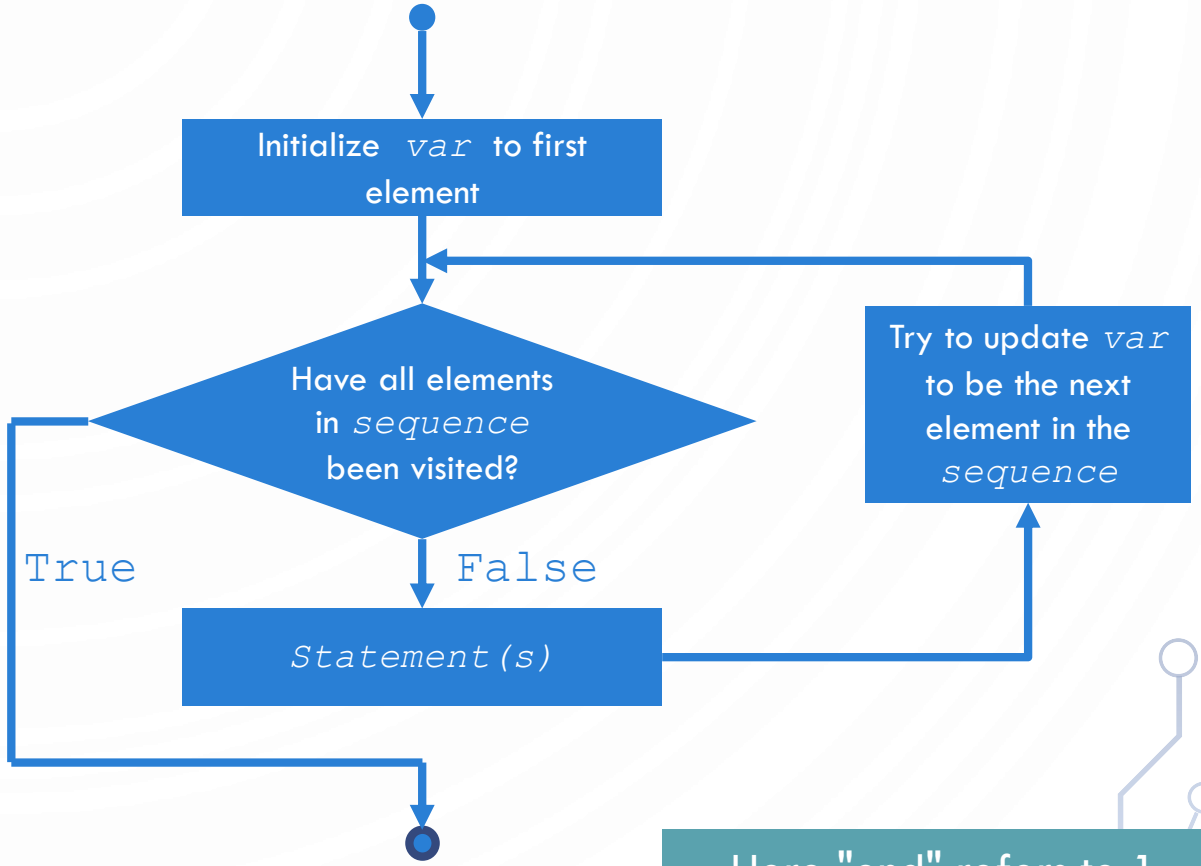

#include <stdio.h>
int main(void)
{
int count;
for (count = 1; count <= 500; count++)
printf ("I will not throw paper airplanes in class.");
return 0;
}
NICE TRY.

FOR LOOPS

```
1.    for var in sequence:
2.        # loop body
3.        Statement(s)
```

Example

```
1.    for x in range(0, 100):
2.      print("Welcome to Python!")
```



Here "end" refers to 1 after the last element of the sequence.

TRACING FOR LOOPS

```
1. for x in range(0, 2):
2.     print("Welcome to Python!")
```

Memory	Output

TRACING FOR LOOPS

```
1.  for x in range(0, 2):
2.      print("Welcome to Python!")
```

Initialize x

Memory
x: 0

**Note* range(0, 2) is [0, 1]*

Output

TRACING FOR LOOPS

```
1.  for x in range(0, 2):
2.      print("Welcome to Python!")
```

Have all elements been visited? No

Memory
x: 0

**Note* range(0, 2) is [0, 1]*

Output

TRACING FOR LOOPS

```
1. for x in range(0, 2):
2.     print("Welcome to Python!")
```

Output

Memory
x: 0

**Note* range(0, 2) is [0, 1]*

Output
Welcome to Python!

TRACING FOR LOOPS

```
1.  for x in range(0, 2):
2.      print("Welcome to Python!")
```

Try to set x to next element of sequence

Memory
x: ~~0~~ 1

**Note* range(0, 2) is [0, 1]*

Output
Welcome to Python!

TRACING FOR LOOPS

```
1.  for x in range(0, 2):
2.      print("Welcome to Python!")
```

Have all elements been visited? No

Memory

x: ~~0~~ 1

**Note* range(0, 2) is [0, 1]*

Output

Welcome to Python!

TRACING FOR LOOPS

```
1.  for x in range(0, 2):
2.      print("Welcome to Python!")
```

Output

Memory
x: ~~0~~ 1

**Note* range(0, 2) is [0, 1]*

Output
Welcome to Python!
Welcome to Python!

TRACING FOR LOOPS

```
1.  for x in range(0, 2):
2.      print("Welcome to Python!")
```

Try to set x to next element of sequence

Memory
x: ~~0~~ 1

**Note* range(0, 2) is [0, 1]*

Output
Welcome to Python!
Welcome to Python!

TRACING FOR LOOPS

```
1.  for x in range(0, 2):
2.      print("Welcome to Python!")
```

Have all elements been visited? Yes

Memory

x: ~~0~~ 1

**Note* range(0, 2) is [0, 1]*

Output

Welcome to Python!
Welcome to Python!

TRACING FOR LOOPS

```
1.  for x in range(0, 2):
2.      print("Welcome to Python!")
```

Memory
x: ~~0~~ 1

**Note* range(0, 2) is [0, 1]*

Output
Welcome to Python!
Welcome to Python!

RANGE

- Range is a method that generates a sequence of integer numbers
  - `range(a, b, step)` – generates numbers from a up to but not including b with an increment of step, e.g., `range(2, 10, 2)` returns $[2, 4, 6, 8]$
  - `range(a, b)` – generates numbers from a up to but not including b with an increment of 1, e.g., `range(1, 5)` returns $[1, 2, 3, 4]$
  - `range(b)` – generates numbers between 0 and b with an increment of 1, e.g., `range(3)` returns $[0, 1, 2]$

PRACTICE

- Group 1: Write a for loop to output all numbers between integers a and b
- Group 2: Write a for loop to output the multiples of an integer a up to N
- Group 3: Write a for loop to output all the even numbers from 100 to 999 in reverse order.

COMPARE FOR LOOPS TO WHILE LOOP

```
1.  count = 0
2.  while count < 100:
3.    print("Welcome to Python")
4.    count += 1
```

```
1.  for x in range(1, 100):
2.    print("Welcome to Python!")
```

Note, each has their own use.
For loops are a special case in which each element of a sequence is visited. In this case (and only this case) are for-loops appropriate in Python.

NESTING

- In control flow, nesting is where you place a control structure inside of another
- Example: 2 for loops to print a multiplication table

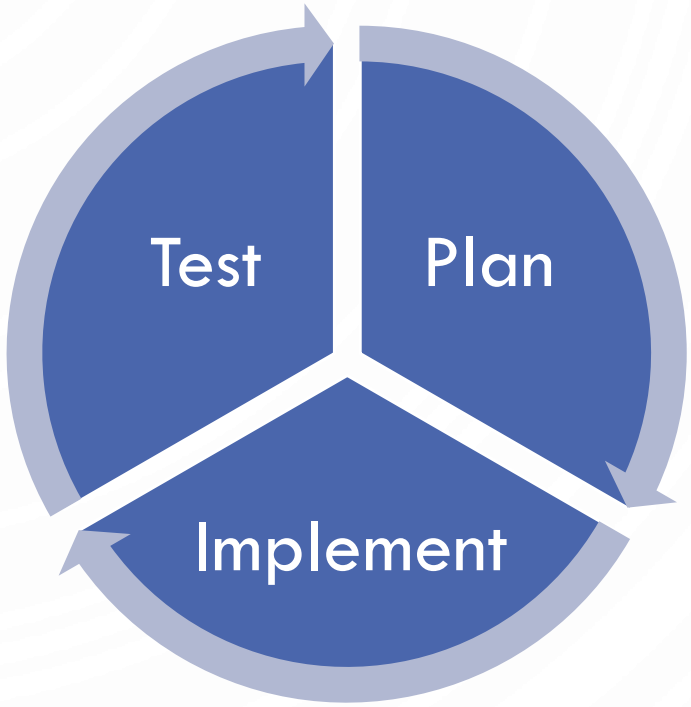
```
1.  for i in range(0, 10):
2.      for j in range(0, 10):
3.          print(str(i) + "*" + str(j) + " = "
                  + format(i*j, "2d"), end="   ")
4.      print() # Print a new line
```

EXERCISE – FIX THE GUESSING GAME

- Lets fix our guessing game program to allow up to 20 guesses. Additionally, try to protect against bad input
- Program this together
- If you get lost program is on following slides (split into multiple slides)

EXERCISE – WHERE TO BEGIN

- When developing programs
  - Always think first!
  - Sketch out solution, i.e., plan
  - Implement solution
  - Test solution
  - Repeat!
- Called iterative development



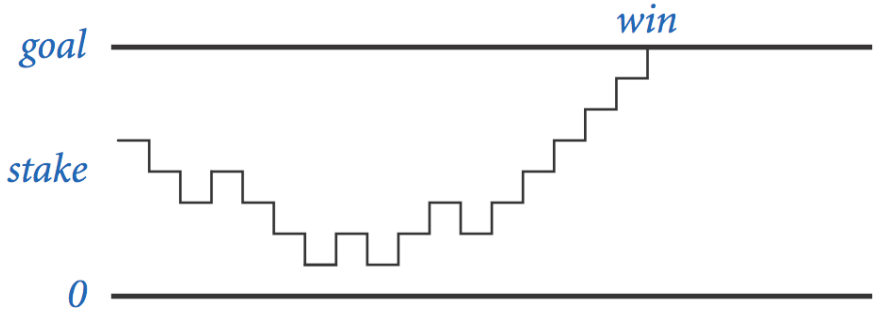
EXERCISE – FIX THE GUESSING GAME

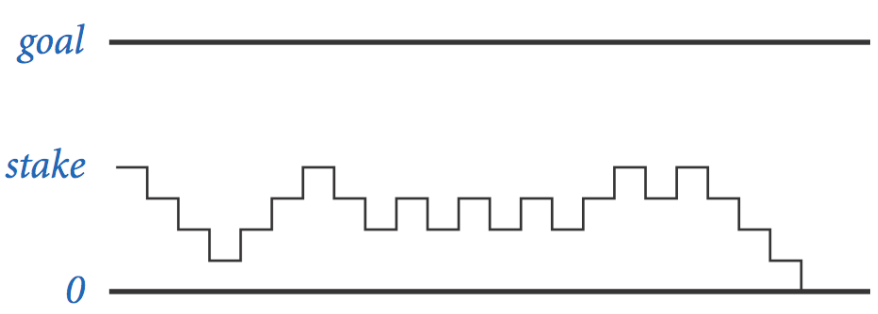
```
1.    import random
2.
3.    # Grab a random number
4.    rn = random.randint(1, 99)
5.    un = 0
6.    guesses = 0
7.
8.    # Allow user to continually guess
9.    while rn != un and guesses < 20:
10.     un = int(input("Please enter a
            number between 1 and 99: "))
11.
12.     if un < 1 or un > 99:
13.       print("Invalid guess.")
14.     elif un == rn:
15.       print("You won!")
16.     elif un > rn:
17.       print("Too high")
18.       guesses += 1
19.     else: # un < rn
20.       print("Too low")
21.       guesses += 1
22.
23.  if guesses == 20:
24.    print("You lost. Out of
         guesses. The correct number
         is " + str(rn) + ".")
```

MONTE CARLO SIMULATION

GAMBLER'S RUIN

- Gambler's ruin. Gambler starts with $stake and places $1 fair bets until going broke or reaching $goal.
  - What are the chances of winning?
  - How many bets will it take?
- One approach. Monte Carlo simulation.
  - Flip digital coins and see what happens.
  - Repeat and compute statistics.





GAMBLER'S RUIN

```
1.   import random
2.
3.   stake, goal, T = eval(input("Enter stakes, goal, and T: "))
4.
5.   wins = 0
6.   for t in range(T):
7.     cash = stake
8.     while cash > 0 and cash < goal:
9.       if random.random() < 0.5:
10.        cash += 1
11.      else:
12.        cash -= 1
13.    if cash == goal:
14.      wins += 1
15.
16.  print(wins, "wins of", T)
```

```
% python3 Gambler.py 5 25 1000
191 wins of 1000

% python3 Gambler.py 5 25 1000
203 wins of 1000

% python3 Gambler 500 2500 1000
197 wins of 1000
```

OTHER CONTROL FLOW STATEMENTS

OTHER HELPFUL STATEMENTS FOR LOOPS

- **break** – immediately exit the loop. Do not continue executing any more of the loop. Example:

```
while True:
    if q-key-is-pressed():
        # quit the game
        break
    Game-loop()
```

- **continue** – immediately skip to the end of the body of the loop, i.e., start next iteration. Example:

```
for i in range(0, 10):
    if(isPrime(i))
        # OCD against prime numbers
        continue
    HandleNotPrimes()
```

CONTROL FLOW SUMMARY

- Control flow.
  - Sequence of statements that are actually executed in a program.
  - Conditionals and loops: enable us to choreograph the control flow.

Control Flow	Description	Examples
Straight-line programs	All statements are executed in the order given	
Conditionals	Certain statements are executed depending on the values of certain variables	`if; if-else`
Loops	Certain statements are executed repeatedly until certain conditions are met	`while; for`

EXERCISE

- Write a program to draw a checkerboard pattern with Turtle (either a Checker's board or a Chess board)
  - You can set the speed of the turtle to infinity (`turtle.speed(0)`)
  - Turtle allows the ability to draw a filled rectangle with `turtle.begin_fill()` and `turtle.end_fill()`