Lecture 3 - `while` and `if` statements

Computer Programming

Robert D. Vincent and Samia Hilal

Marianopolis College

February 6, 2022

Loops in Python

- Computers are used to automate repetitive tasks.
- A loop is a control structure used to repeat a given section of code.
- A loop runs (executes) for 0 or more `iterations`.
- Loops are very basic but also powerful programming concepts.
- In Python, we will be looking at 2 different loop structures based on how each loop continues:
  - `The while loop`: Continues until a particular condition is met.
  - `The for loop`: Continues for a certain number of times.

The `while` loop

- The Python `while` loop has the form:

```
while expression : # Colon is required.
    statement1     # Statement list
    statement2
    ...
```

- The statement list is the *body* of the loop.
- The Boolean expression is sometimes called the *Condition* or *Test Expression*.

The `while` loop

The Python `while` loop can be represented:

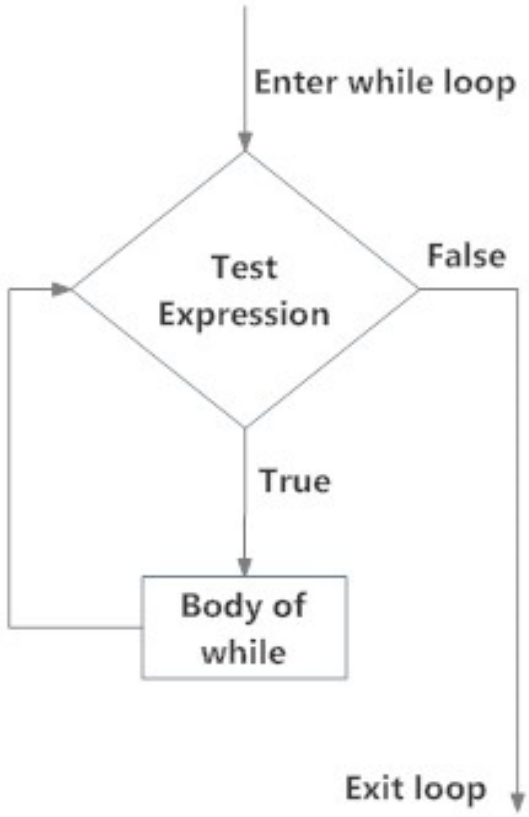


Fig: operation of while loop

How it works

1. Evaluate the condition. If it is `False`, go to 4.
2. Run the loop body.
3. Go back to 1.
4. Continue with next statement.

Two important points:

- The body might never be executed if the condition is `False`.
- The condition is re-evaluated **once** each time through the loop.

A first example

```
#simple-while.py

count = 0
while (count < 9):
    print ('The count is:', count)
    count = count + 1

print "Good bye!"
```

Compound statements

- These three statements are simple:

```
a = 0
print(a)
a += 1
```

- The `while` loop is type of *compound* statement:
- The entire loop (header and body) is a single compound statement.
- For each compound statement, header lines end in a colon. This introduces the indented statement list.

A Manual Algorithm Example

- Find the square root of a number x:
  1. Choose an initial square root guess, y
  2. Is $y \cdot y$ is "close enough" to x?
     - *Yes:* Then the answer is y. Stop
     - *No:* Go to step 3.
  3. Set y equal to the average of y and x/y.
  4. Repeat from step 2 using the new value for y.
- Example with $x = 3$, initial guess $y = 2$.

y	$y \cdot y$	$y = (y + x/y)/2$
2	4	1.75
1.75	3.0625	1.73214
1.73214	3.00032	1.73205
1.73205	3.00000	1.73205

Elements of an algorithm

- A sequence of simple operations.
- A *flow of control* that specifies the order in which the operations are performed (or *executed*).
- Some means to determine when to stop.

Automating previous example

- Recall our square root algorithm.
- It repeats while our guess is not "close enough."
- A good application of the `while` loop.
  - Our condition must check whether our guess is close enough to the true square root.
  - Our body must run the update step.

```
#l3sqrt.py

x = 3 # Input number.
y = 1 # Initial guess.
# Repeat while |y*y - x| > 1e-10
while abs(y * y - x) > 1e-10:
    y = (y + x / y) / 2 # Update
print('The square root of', x, 'is', y)
```

Indentation in Python

- Python is one of the few languages that enforces neat indentation!
- Every statement in the body of a complex statement *must* be indented.
- At least one indented statement must be given.
- The first statement that is **not** indented is the "next" statement after the complex statement.
- Each indented line must use the same arrangement of spaces and tabs.
- Applies to all compound statements in Python.

Indentation example

```
#l3indent.py

x = int(input("Enter an integer: "))
y = 1
n = x
while n > 0:
    y *= n    # first looping statement
    n -= 1    # second looping statement
# This statement follows the loop!
print(x, 'factorial is', y)
```

Common indentation problems

```
#l3error.py

n = 1
while n < 10:
n += 1          # ERROR: Forgot indent.

n, r = 10, 1
while n > 0:
    r *= n
   n -= 1       # ERROR: Reduced indent.

x, y = 1, 1
while x < n:
    y *= x
      x += 1 # ERROR: Extra indent.
```

Nested loops

- Loops can be *nested:* l3nest1.py

```
i = 0          # outer initialization
while i < 3:
   print("i =", i)
   j = 0       # inner initialization
   while j < 3:
      print("j =", j)
      j += 1   # inner increment
   i += 1      # outer increment
```

- The *inner* loop runs 3 times for each repetition of the *outer* loop.
- The inner loop is indented relative to the outer loop.

Nested loop example

```
#l3nest2.py

# Compute 'triangular' numbers.
msg = 'Enter a number or q to quit: '
string = input(msg)
while string != 'q':
    number = int(string)
    total = 0
    counter = 1
    while counter <= number:
        total += counter
        counter += 1
    print("The total of numbers 1 to",
          number, "is", total)
    string = input(msg)
print("Finished.")
```

The `if` statement

- The `while` statement expresses repetition.
- The `if` statement expresses conditional or alternative actions:

```
if expression :
    statement1
    statement2
    ...
```

- If the `Test Expression` is `True`, execute the statement list (`Body of if`). Otherwise continue with the next statement after the indented list.

The `if` statement

The **if** statement can be expressed graphically as follows:

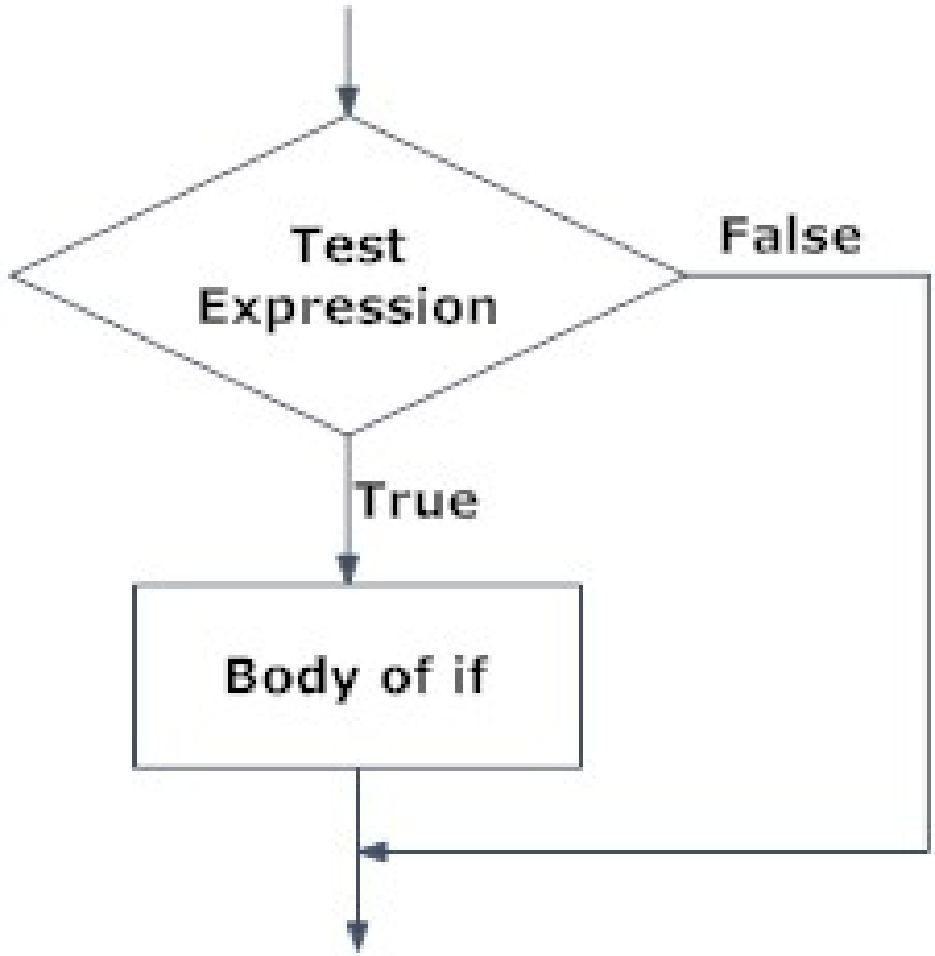


Fig: Operation of if statement

Example

- Given the program (l3if1.py):

```
x = 1
if x < 2:          # 1st if statement
    print("A")
if x < 1:          # 2nd if statement
    print("B")
print("C")
```

- This would print:

```
A
C
```

- Each `if` reserved word introduces a new `if` statement!

The `else` clause

- An `if` statement may have an `else` part:

```
if expression :
    statement1
    statement2
    ...
else:
    statement1
    statement2
    ...
```

- If the first expression is `True`, execute the first statement list. Otherwise, execute the statement list after the `else`.

The `else` clause

The **if** statement and its **associated else** can be pictured as follows:

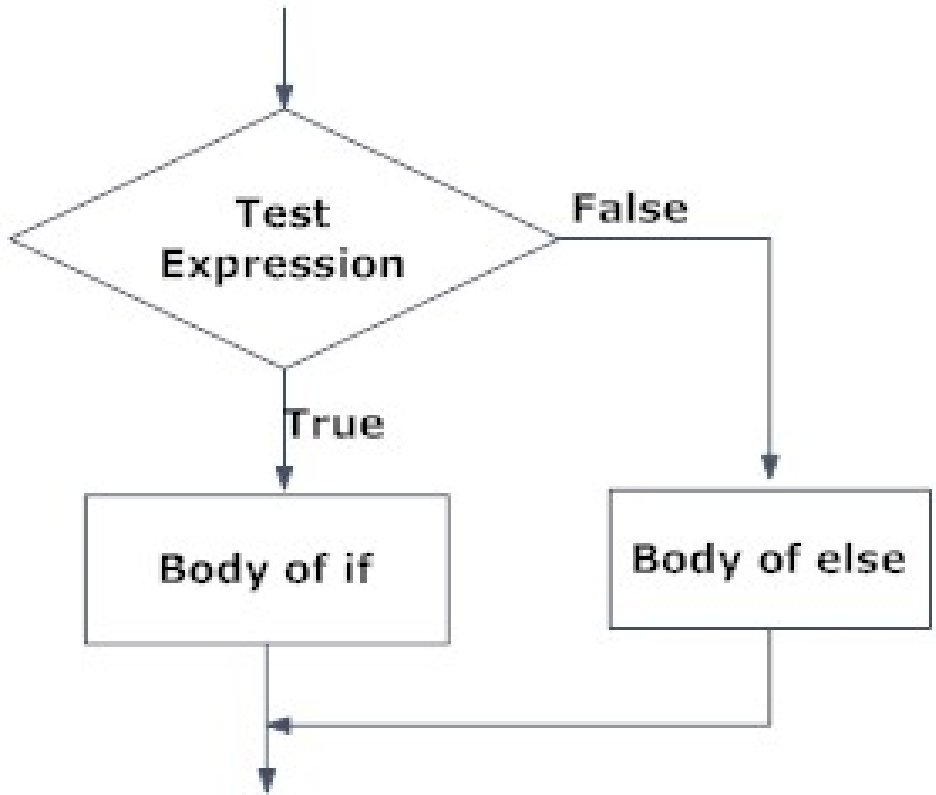


Fig: Operation of if...else statement

The `elif` clause

- An `if` statement may have several `elif` parts:

```
if expression :
    statement1
    statement2
    ...
elif expression :
    statement1
    statement2
    ...
```

- If the first expression is `True`, execute the first statement list. Otherwise, if the second expression is `True`, execute the second statement list.

The `if elif else` Together

An `if` statement with `one elif` part and `an else` clause can be seen here:

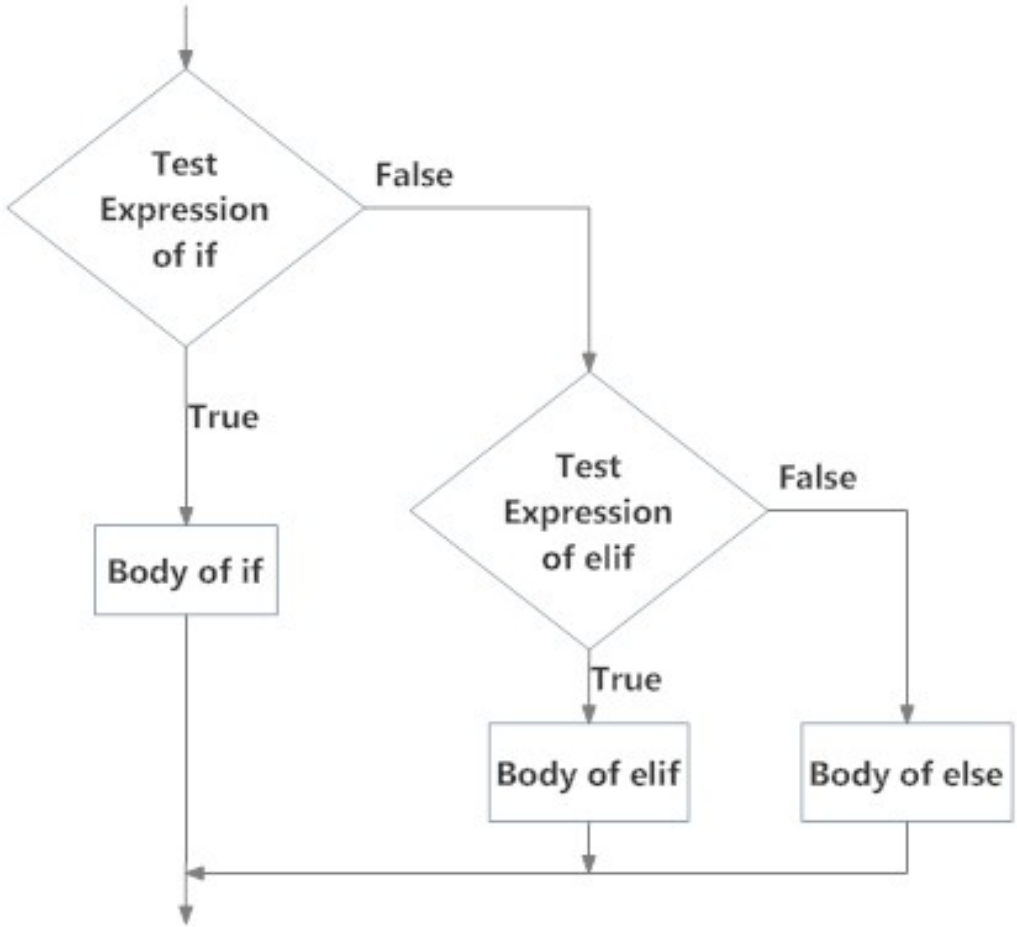


Fig: Operation of if...elif...else statement

Fitting the pieces together

- The `if` section goes first.
- There may be zero or more `elif` sections.
- Finally, at most one `else` section.
- At *most* one of the statement lists will execute.
- Example: l3if2.py

```
x = 5
if x < 2:
    print('tiny')
elif x < 6:
    print('small') # only this prints!
elif x < 10:
    print('medium')
else:
    print('large')
```

The break statement

- break exits the nearest enclosing loop.
- Example: l3brk.py

```
x = float(input('Enter a number: '))
y = 1 # Initial guess.
n_guesses = 0
error = 1e-5
while abs(y * y - x) > error:
  n_guesses += 1
  if n_guesses > 1000: # Limit our guesses!
    break
  y = (y + x / y) / 2  # Update
print('Found', y, 'in', n_guesses, 'guesses')
```

The `continue` statement

- `continue` skips to the next iteration of the loop
- Example: l3cont-evens.py

```
# Print even numbers from 1 to n
n=int(input("Enter a positive int>0:"))
number = 0
while (number <= n):
    number = number + 1
    if (number % 2) != 0:
        continue
    print(number)
```

- In effect, it jumps back to the "top" of the loop.

Algorithms

- Recall the algorithm for square root we introduced at the beginning of the course.
- It is one example of a "guess and check" algorithm:
  1. Generate an initial **guess**.
  2. **Check** the guess, finish if correct.
  3. Improve the **guess**.
  4. Return to 2.
- These are often "approximation" algorithms, because computing the exact answer may be impossible or impractical.
- The choice of algorithm has a profound effect on performance.

Algorithm 1 - Exhaustive enumeration

- "Brute force" method:
  1. Start at zero (assuming a positive answer).
  2. Check if close enough to answer.
  3. Increment by some tiny amount.
  4. Go to 2.
- Very slow.
- Hard to get "close" to the right answer.

Algorithm 1 in Python

```
#l3sqrt1.py

# Exhaustive enumeration - search for the
# square root by starting at zero and
# counting up.
x = float(input('Enter a number: '))
tiny = 1e-6
error = 1e-5
y = tiny
n_guesses = 0
while abs(y * y - x) > error:
    n_guesses += 1          # count guesses!
    y += tiny
print('Found', y, 'in', n_guesses, 'guesses')
```

Instrumentation in code

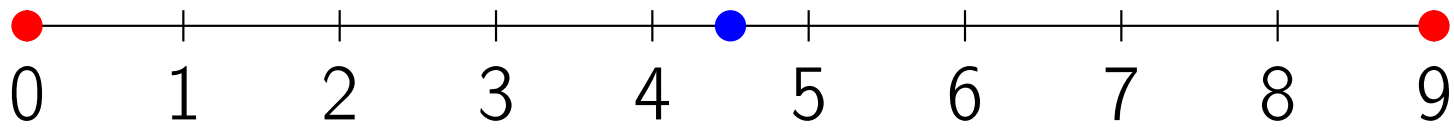
- Notice the use of `n_guesses`.
- This is not *necessary* for this calculation.
- What does it do?
  - Counts number of times the loop repeats.
  - Allows us to assess program behavior, apart from the required calculations.
- This is sometimes called *instrumented* code.
- Using code to measure performance and error results.

Algorithm 2 - Bisection search

- Divide and conquer:
  1. Start in the middle of the possible range.
  2. Check if close enough to the right answer.
  3. If not, cut the range in half:
     - If the guess is too low, use the upper half.
     - If the guess is too high, use the lower half.
  4. Choose a new guess to be the midpoint of the revised range.
  5. Go to 2.
- Much more efficient.
- Works only for monotonic functions.

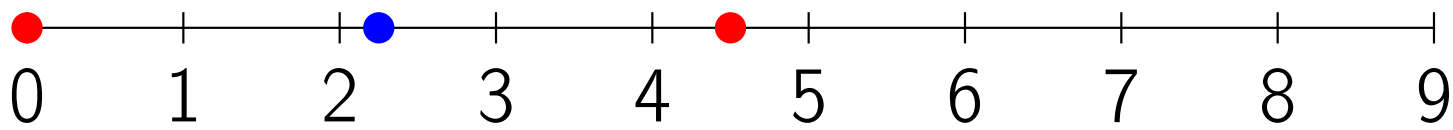
Bisection search example 1

- Suppose I want to find the square root of 9.
  - I start the range from $lo = 0$ to $hi = 9$
  - Midpoint is 4.5, but $4.5^2 = 20.25$ is *too big*.
  - So make $hi = 4.5$ and repeat.



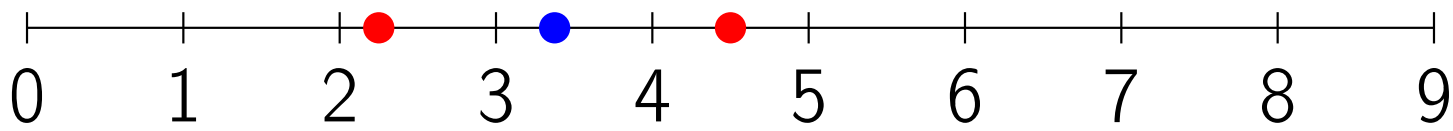
Bisection search example 2

- Now $lo = 0$ to $hi = 4.5$
- Midpoint is now 2.25, which is *too small*.
- So make $lo = 2.25$ and repeat.



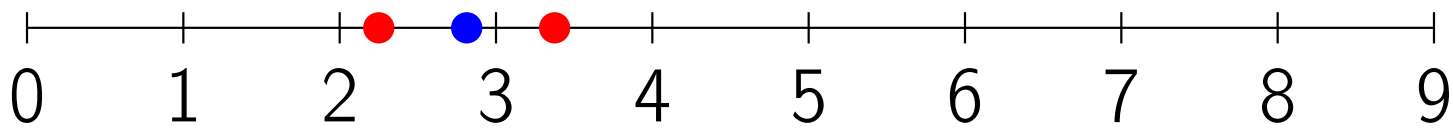
Bisection search example 3

- Now $lo = 2.25$ to $hi = 4.5$
- Midpoint is now 3.375, which is too large.
- So make $hi = 3.375$ and repeat.



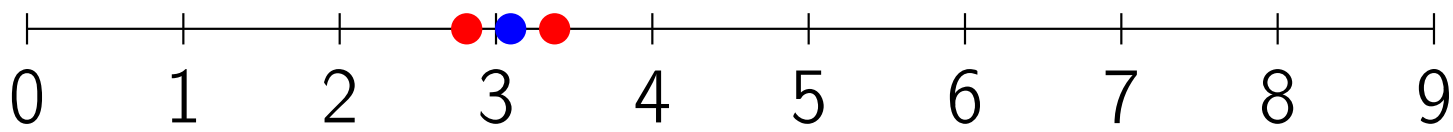
Bisection search example 4

- Now $lo = 2.25$ to $hi = 3.375$
- Midpoint is now 2.8125, which is too small.
- So make $lo = 2.8125$ and repeat.



Bisection search example 5

- Now $lo = 2.8125$ to $hi = 3.375$
- Midpoint is now 3.09375, which is close enough!



Algorithm 2 in Python

```
# Bisection search. Divide the search area
# in half on each guess. (l3sqrt2.py)
x = float(input('Enter a number: '))
lo, hi = 0, x # double assignment!
error = 1e-5
y = (lo + hi) / 2
n_guesses = 0
while abs(y * y - x) > error:
    n_guesses += 1
    if y * y > x:
        hi = y  # lower half
    else:
        lo = y   # upper half
    y = (lo + hi) / 2
print('Found', y, 'in', n_guesses, 'guesses')
```

Performance

- For the square root of 2:
  - Algorithm 1 - 1,414,210 guesses.
  - Algorithm 2 - 15 guesses.
- For the square root of 100:
  - Algorithm 1 - 9,999,999 guesses.
  - Algorithm 2 - 26 guesses.
- Different algorithms can have very different performance!

Summary

- We learned two *control structures*, the `while` and `if` statements.
- `while` and `if` are both compound statements.
- `break` and `continue` are simple statements that modify the behavior of loops.
- We examined several square root algorithms and saw how they used these control structures.
- The key point about algorithms: choice of algorithm matters.