Introduction to Programming

Dr. John Stavrakakis

School of Computer Science, University of Sydney

Copyright Warning

Week 3: Control Flow 1: Branching and Looping

We will cover: Branching with `if` and `else`, Looping with `while` and more on handling numbers: `if`/`else`, `switch`

You should read: §§1.2 and 1.3 of Sedgewick

Lecture 5: Control Flow 1: `if`

Handling simple choices

Evaluating multiple conditions

Boolean expressions

Boolean logic

Operation	*Meaning*
x and y	true if both x and y are true, false otherwise
x or y	true if both x or y or both are true, false otherwise
not x	true if x is false
x == y	true if x and y are both true or both false, false otherwise
x != y	true if x is true and y is false, or x is false, false otherwise

Complex boolean expressions

Ask a simple question that has a yes/no answer.

Can depend on sub problems to be solve first.

Question: Ready to go out?

depends on: have keys? have phone? if it is raining, do I have umbrella?

```
have_keys = True
have_phone = True
is_raining = True
have_umbrella = False

ready_to_leave = ?
```

What are all the cases is readyToLeave `True` or `False`. Use your truth tables!

Complex boolean expressions

Question: Ready to go out?

depends on: have keys? have phone? if it is raining, do I have umbrella?

```
readyToLeave = ( have_keys and have_phone ) and \
   ( ( is_raining and have_umbrella ) or ( not is_raining ) )
```

Complex boolean expressions

What is the boolean result?

```python
# minimum needed for cake
# 1/2 cup butter
# 3/4 cup white sugar
# 1/2 cup cocoa powder
# 3 eggs
# 1 teaspoon vanilla extract
# 4 squares chopped chocolate (optional)

can_make_cake = ???

print("can_make_cake = " + str(can_make_cake))
```

Pay careful attention to the specification first, then derive the code.

We need to be careful to evaluate the expressions as intended.

Casting

Treating variables as other types

Casting primitive

Operators used in an expression can be evaluated differently based on the data types involved.

What is the result?

```
litres = 2
portion = 0.330
persons = litres / portion
print("invite " + str(persons) + " persons")
```

casting allows the programmer to explicitly tell the compiler to treat a variable, or expression, as another type.

Casting primitive types

Convert integer to floating point number

```python
litres = 2
portion = 0.330
persons = int(litres / portion) # change the type
print("invite " + str(persons) + " persons")
```

The opposite is also possible. What is the result?

```python
litres = float(2)
persons = float(5)
portion = litres / persons
print("each person drinks " + str(portion) + "mL")
```

Can we cast in different places, where is the best?

if

(AN UNMATCHED LEFT PARENTHESIS CREATES AN UNRESOLVED TENSION THAT WILL STAY WITH YOU ALL DAY.

Source: xkcd

The if statement

- What's it for?
    - to have different behaviour in a program based on whether something is true or false (this is a kind of *control statement*).
- What does it look like?

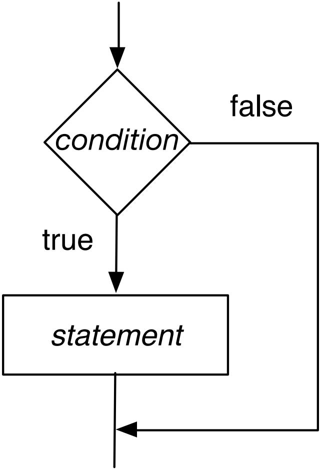
```
if condition :
    statement
```

- *condition* is a boolean expression that evaluates to `True` or `False`
- *statement* is one computer instruction or a sequence of computer instructions within the same indentation block
- In code this looks like

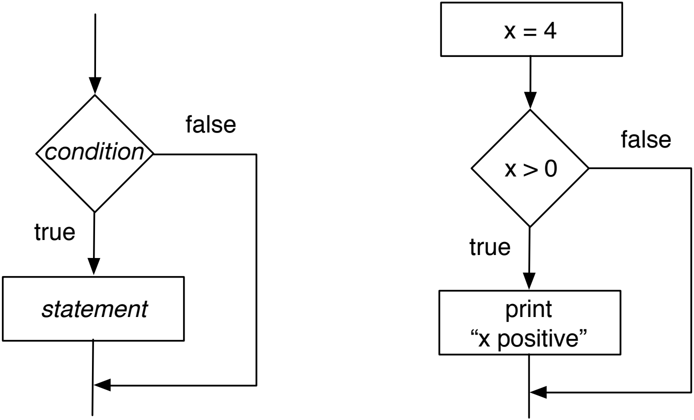
```
if x > 0 :
   print("x is positive")
```

How `if` works

The “if” statement is a simple control flow structure: it is used to test the value of an expression, to see whether it’s true, and if true, then to do something else.

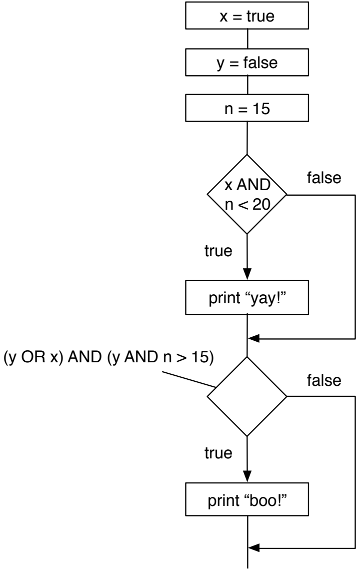


How `if` works


condition
false
true
statement
x = 4
x > 0
false
true
print
“x positive”

More logic with `if`

The *condition* can be a complex boolean expression. What is the code?



x = true

y = false

n = 15

x AND n < 20

false

true

print “yay!”

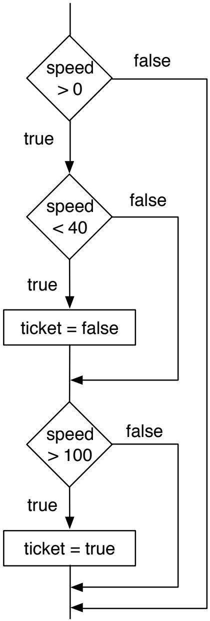
(y OR x) AND (y AND n > 15)

false

true

print “boo!”

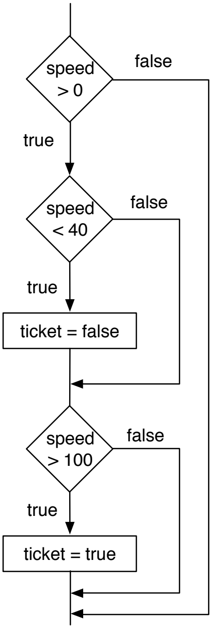
Nested if



You can put ifs inside ifs, like this:

```
ticket = False
if speed > 0:
  if speed < 40:
    ticket = False
  if speed > 100:
    ticket = True
```

Deskcheck



Control statements like `if` can lead to different statements being executed, code paths.

The variables will have different values depending on the current conditions

speed	ticket
0	F

Extending `if`

`if` you had to write an "`if`" for each possible outcome you can easily miss a case:

```
      if x > 0 :
         # do something

      if x <= 0 :
         # do something else
```

what is the opposite case?

```
if (i == 7 and j < i) or \
   ( (not ((j - i) < 5)) and (j != 0) ):
   # do something

if (i != 7 or j >= i))) and \
   ( (not ((j - i) >= 5)) or (j == 0) ) ):
   # do something else
```

Extending if (cont.)

if you had to write an “if” for each possible outcome you can easily miss a case:

```
      if x > 0 :
         # do something

      if not (x > 0) :
         # do something else
```

```
if (i == 7 and j < i) or \
   ( (not ((j - i) < 5)) and (j != 0) ):
   # do something

if not ((i == 7 and j < i) or \
   ( (not ((j - i) < 5)) and (j != 0) ) ):
   # do something else
```

Extending `if` (cont.)

The keyword `pass` can help make an *empty* statement. It does NOTHING, but it is a syntactically correct statement that we can place

```
      if x > 0 :
         pass
```

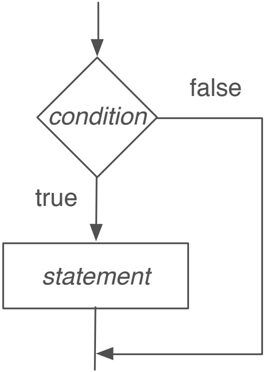
Why? Simplicity and readability.

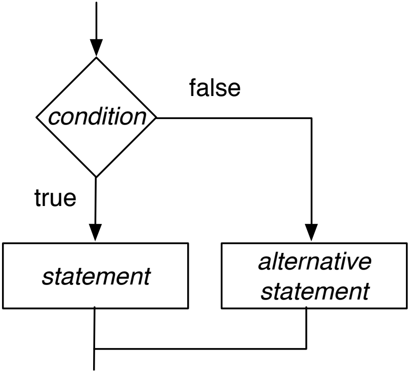
```
if (i == 7 and j < i) or \
   ( (not ((j - i) < 5)) and (j != 0) ):
   # do nothing when condition is True
   pass
else:
   # do something when condition is False
```

`if ... else`

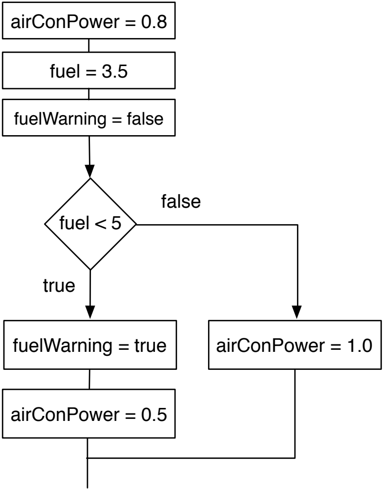
The alternative to writing both cases out explicitly is to use `else`. Here's the syntax:

```
if condition then
    Statements
else
    AlternativeStatements
end if
```





Example



airConPower = 0.8

fuel = 3.5

fuelWarning = false

fuel < 5

false

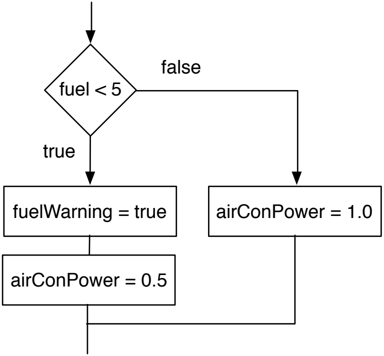
true

fuelWarning = true

airConPower = 1.0

airConPower = 0.5

Deskcheck 2



Don’t always know values ahead of time

Use desk check to track each variable value and how it changes

As part of testing, you should pick values that are normal, abnormal, on the boundary etc.

	fuel	airConPower	fuelWarning	
	0			
	5			

if ... elif ... elif ...

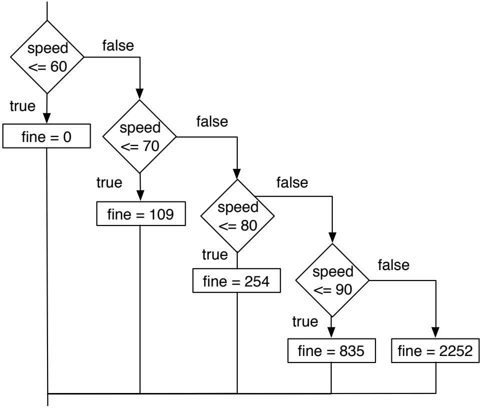
You can have separate conditions checked in sequence. `if` statements with the alternative statement as `elif` (short for else if):

```python
if ch == 'a':
  # do thing 1
elif ch == 'b':
  # do thing 2
elif ch == 'c':
  # do thing 3
```

It is equivalent to putting the expressions after the `else`s into their own separate blocks.

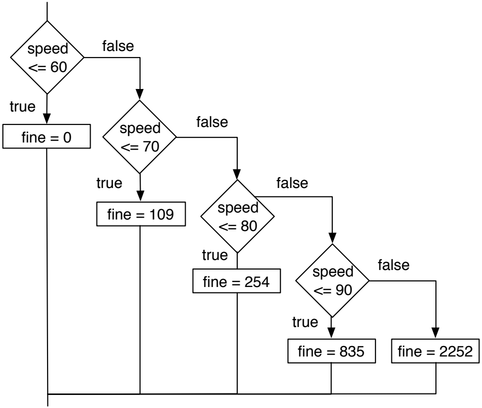
if ... elif ... elif ...

An example of checking conditions in a specific order



```
if speed <= 60:
  fine = 0
elif speed <= 70:
  fine = 109
elif speed <= 80:
  fine = 254
elif speed <= 90:
  fine = 835
else:
  fine = 2252
```

Deskcheck 3



speed	fine
-789	