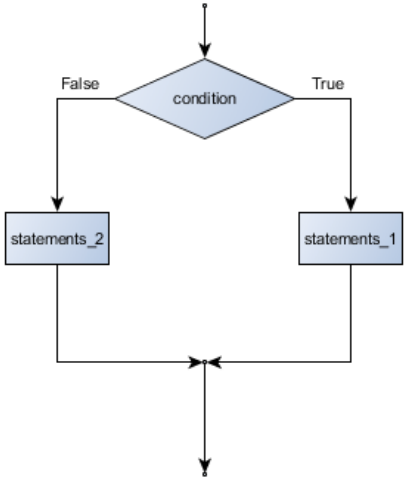
Conditionals



CS111 Computer Programming

Department of Computer Science
Wellesley College

Overview: Making Decisions

If "is it raining":

take the umbrella

wear rainboots

wear raincoat

Else:

wear sandals

wear a summer dress

"Is it raining" is an expression that can return True or False.

In a Python program we can use:

- True/False values
- Relational Expressions
- Logical Expressions
- Predicates

(all evaluate to True/False) whenever the code needs to make a decision for what to do next.

Conditionals (`if` Statements)

Concepts in this slide: Conditional statement syntax involving `if` and `else`.

Boolean expressions are used to choose between two courses of action in a conditional statement introduced by the keyword `if`.

```
if boolean_expression:
    statement1
    statement2
    statement3
else:
    statement4
    statement5
```

```
def abs(n):
    '''returns absolute value'''
    if n < 0:
        return -n
    else:
        return n
```

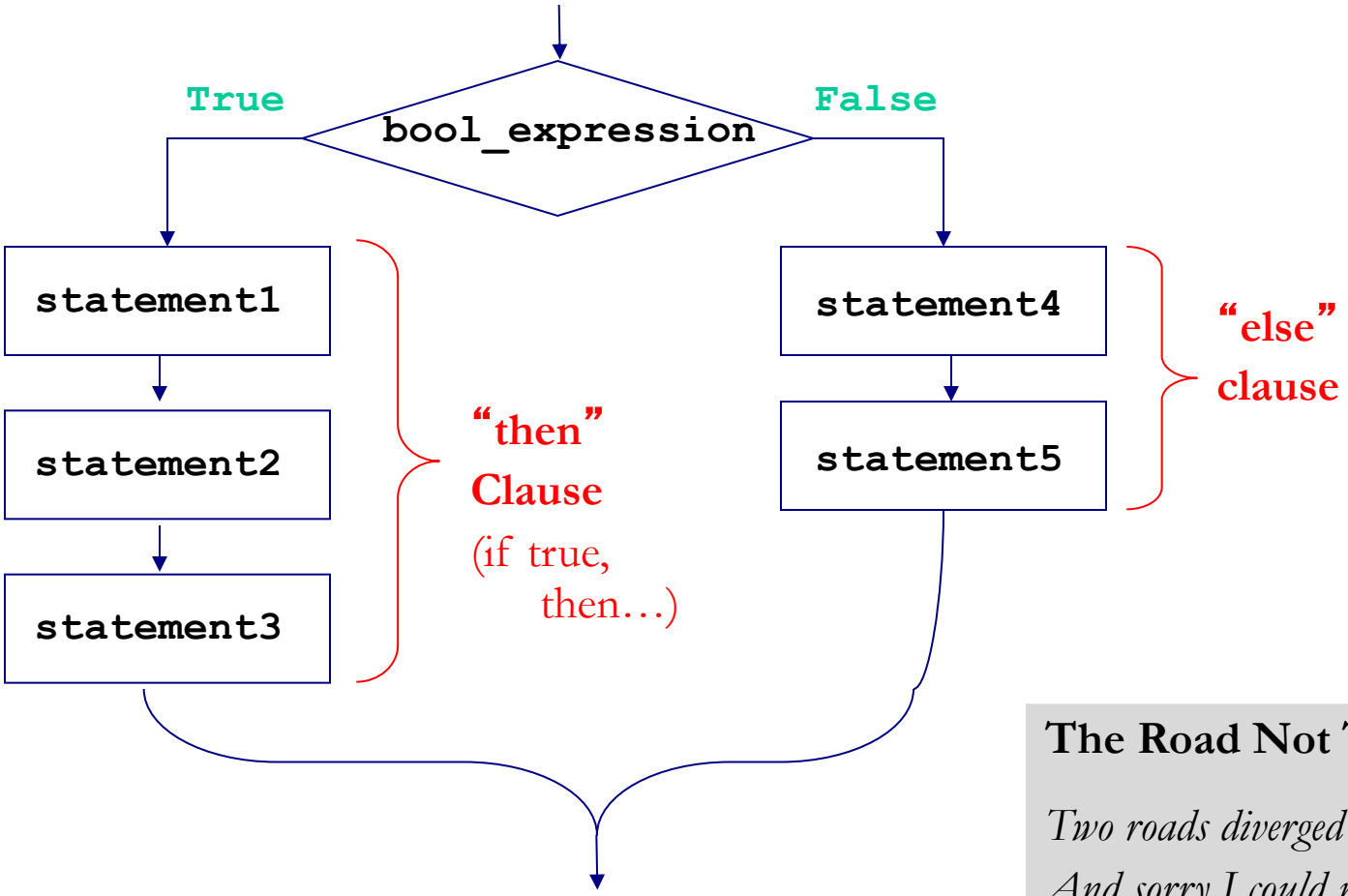
Above is the Python syntax for expressing conditional statements. Notice:

- Colons at the end of line for `if` and `else`
- Indentation for lines succeeding `if` and `else`

Note: "else" clause is optional!

Flow Diagrams

Concepts in this slide: Flow diagrams: a model to understand branched execution.



The Road Not Taken

Two roads diverged in a yellow wood,
And sorry I could not travel both

Robert Frost

IMPORTANT: Only one of the branches is ever executed when a conditional statement is encountered. That is what the Flow Diagram exemplifies.

Expressing the Same Function Two Ways

Are these two functions logically equivalent?
Do they return the same answer for all inputs?

```
def abs(n):
    '''returns absolute value'''
    if n < 0:
        return -n
    else:
        return n
```

```
def abs(n):
    '''returns absolute value'''
    if n < 0:
        return -n
    return n
```

Notice the missing **else**

Nested Conditionals

Concepts in this slide: Syntax for nested conditionals, example of nesting.

```
if boolean_expression1:
    statement1
    statement2
else:
    if boolean_expression2:
        statement3
        statement4
    else:
        statement5
        statement6`
```

```
def movieAge(age):
    if age < 8:
        return 'G'
    else:
        if age < 13:
            return 'PG'
        else:
            if age < 18:
                return 'PG-13'
            else:
                return 'R'
```

A Better Approach: Chained Conditionals

Concepts in this slide: New keyword: `elif`. Replace nesting with chaining of conditionals.

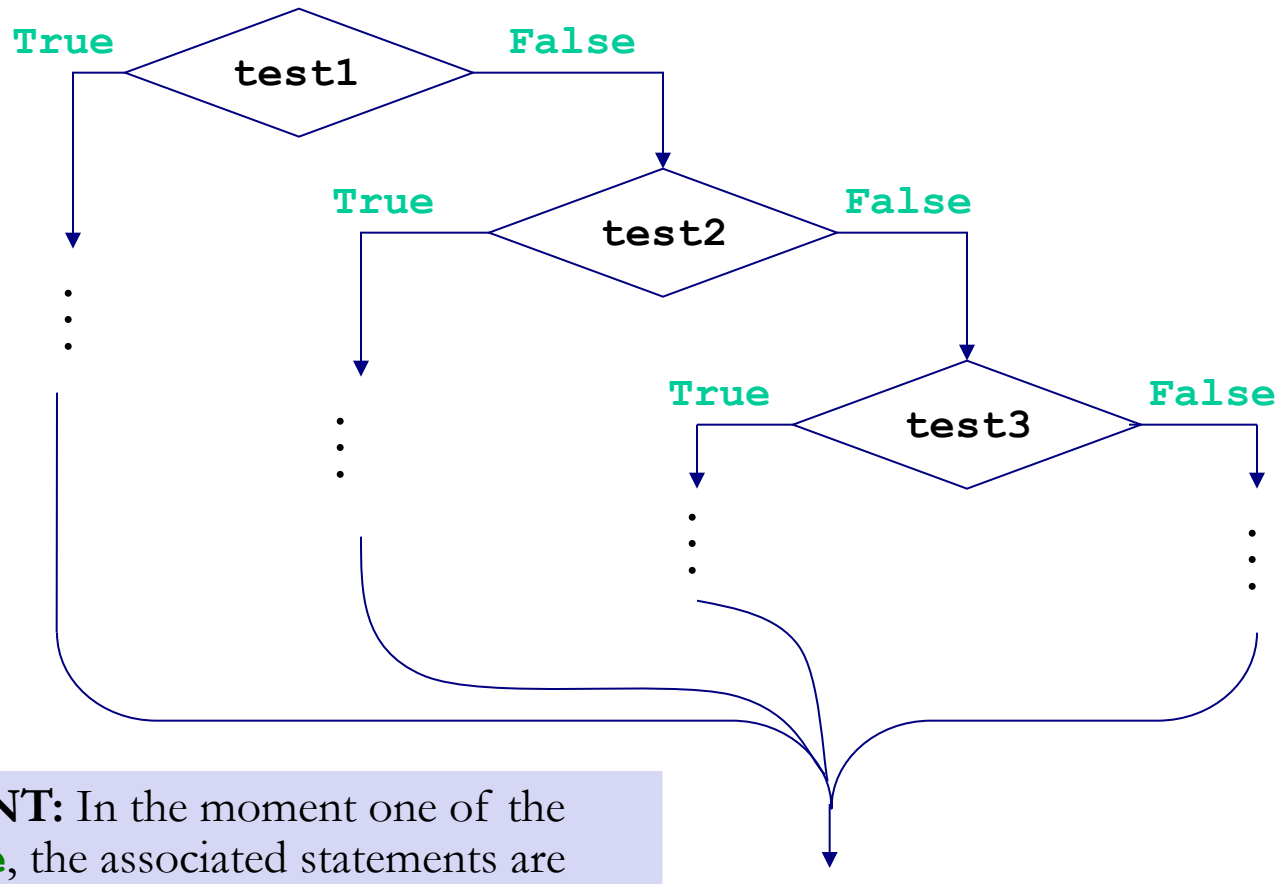
```
if boolean_expression1:
    statement1
    statement2
elif boolean_expression2:
    statement3
    statement4
elif boolean_expression3:
    statement5
    statement6
else:
    statement7
    statement8
```

```
def movieAge(age):
    if age < 8:
        return 'G'
    elif age < 13:
        return 'PG'
    elif age < 18:
        return 'PG-13'
    else:
        return 'R'
```

Compare this implementation of **movieAge** with that of the previous slide. For chained conditionals, we write less code, which is also easier to read because of fewer indentations.

Flow Diagram: Chained Conditionals

Concepts in this slide: Another example of the flow diagram model for branched execution.



IMPORTANT: In the moment one of the tests is `True`, the associated statements are executed and the chained conditional is exited. Only in the case when tests are False, we continue checking to find a True test.

isVowel revisited

The following definition doesn't work. Why?

```
def isVowel(char):
    letter = char.lower()
    return letter == ('a' or 'e' or 'i' or 'o' or 'u')
```

Because by Python's treatment of truthy/falsey values, it's equivalent to

```
def isVowel(char):
    letter = char.lower()
    return letter == 'a'
```

Simplifying Boolean Expressions and Conditionals

Digging Deeper

There are several code patterns involving boolean expressions and conditionals that can be simplified. The unsimplified versions are considered to be bad style and will be flagged by uur Codder tool. Below **BE** stands for any expression evaluating to a boolean, and **STMS** stands for any statements.

Complex Expr/Stmt	Simpler Expr/Stmt	Complex Expr/Stmt	Simpler Expr/Stmt
`BE == True`	`BE`	`BE == False`	`not BE`

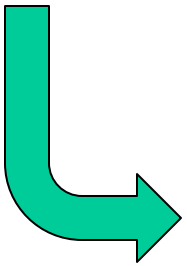
`if BE:`
`  return True`
`else:`
`  return False`
	`return BE`	`if BE:`
		`  return False`
		`else:`
		`  return True`
			`return not BE`

`if BE1:`
`  return BE2`
`else:`
`  return False`
	`return BE1 and BE2`	`if BE1:`
		`  return True`
		`else:`
		`  return BE2`
			`return BE1 or BE2`

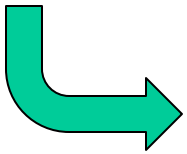
`if BE:`
`  STMS`
`  return True`
`else:`
`  STMS`
`  return False`
	`STMS`
	`return BE`
		`result = BE`
		`return result`
			`return BE`

Simplifying Boolean Expressions and Conditionals: Example


```
def doesNotBeginWithVowel(s):
    if isVowel(s[0]) == False
        return True
    else:
        return False
```



```
def doesNotBeginWithVowel(s):
    if not isVowel(s[0])
        return True
    else:
        return False
```



```
def doesNotBeginWithVowel(s):
    return not isVowel(s[0])
```

All Python values are either Truthy or Falsey

Unexpectedly, in the context of **if**, **and**, and **or**, Python treats a small number of so-called Falsey values (**0**, **''**, **None**, **[]**, **()**, and **{}**) as False and all other values as True (so-called Truthy values).
In general, we think it is bad style to write code that depends on this fact; use Boolean expressions instead!

Digging Deeper

```
def testTruthy(val):
  if val:
    return 'Truthy'
  else:
    return 'Falsey'

testTruthy(True) → 'Truthy'
testTruthy(False) → 'Falsey'
testTruthy(17) → 'Truthy'
testTruthy(0) → 'Falsey'
testTruthy('hello') → 'Truthy'
testTruthy('') → 'Falsey'
testTruthy(None) → 'Falsey'
testTruthy([1,2,3]) → 'Truthy'
testTruthy([]) → 'Falsey'
```

```
0 or 4 → 4
5 or 6 → 5
0 and 7 → 0
8 and 9 → 9
'' or 'a' → 'a'
'b' or 'c' → 'b'
'' and 'd' → ''
'e' and 'f' → 'f'
```