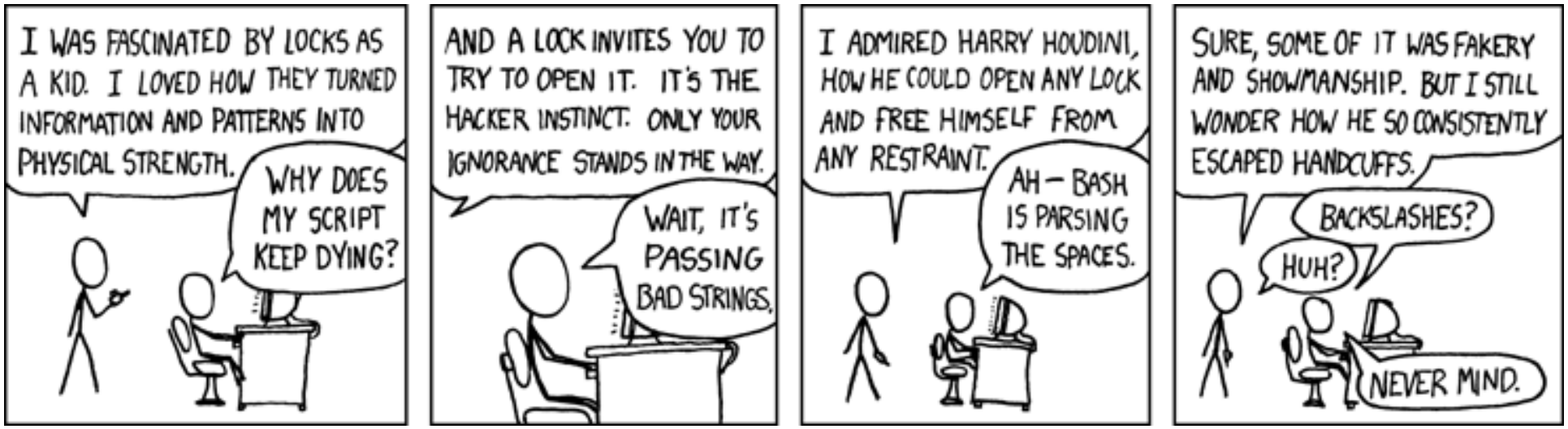
CSc 110, Spring 2018

Lecture 14: Booleans and Strings

Adapted from slides by Marty Stepp and Stuart Reges

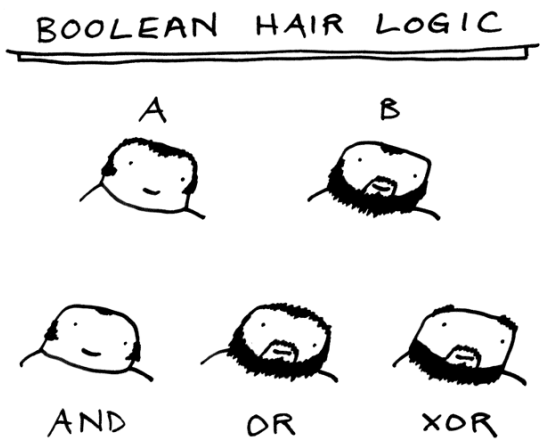


Exercise: Logical questions

- What is the result of each of the following expressions?

```
x = 42
y = 17
z = 25
```

- `y < x and y <= z`
- `x % 2 == y % 2 or x % 2 == z % 2`
- `x <= y + z and x >= y + z`
- `not(x < y and x < z)`
- `(x + y) % 2 == 0 or not((z - y) % 2 == 0)`



  - Answers: `True`, `False`, `True`, `True`, `False`

Type `bool`

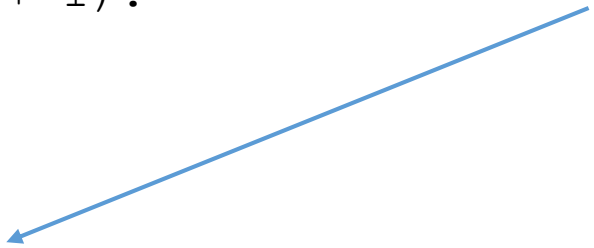
- **`bool`**: A logical type whose values are `True` and `False`.
  - A logical ***test*** is actually a Boolean expression.
  - Like other types, it is legal to:
    - create a `bool` variable
    - pass a `bool` value as a parameter
    - return a `bool` value from function
    - call a function that returns a `bool` and use it as a test

```
minor      = age < 21
is_prof    = "Prof" in name
loves_csc = True

# allow only CS-loving students over 21
if minor or is_prof or not loves_csc:
    print("Can't enter the club!")
```

Returning `bool`

```
def is_prime(n):
    factors = 0;
    for i in range(1, n + 1):
        if (n % i == 0):
            factors += 1

    if factors == 2:
        return True
    else:
        return False
```

Is this good style?



- Calls to functions returning `bool` can be used as tests:

```
if is_prime(57):
    ...
```

"Boolean Zen", part 1

- Students new to `boolean` often test if a result is `True`:

```
if is_prime(57) == True:     # bad
    ...
```

- But this is unnecessary and redundant. Preferred:

```
if is_prime(57):             # good
    ...
```

- A similar pattern can be used for a `False` test:

```
if is_prime(57) == False:    # bad
if not is_prime(57):         # good
```

"Boolean Zen", part 2

- Functions that return `bool` often have an `if/else` that returns `True` or `False`:

```
def both_odd(n1, n2):
    if n1 % 2 != 0 and n2 % 2 != 0:
        return True
    else:
        return False
```

  - But the code above is unnecessarily verbose.

Solution w/ `bool` variable

- We could store the result of the logical test.

```
def both_odd(n1, n2):
    test = (n1 % 2 != 0 and n2 % 2 != 0)
    if test:    # test == True
        return True
    else:       # test == False
        return False
```

- Notice: Whatever `test` is, we want to return that.
  - If `test` is `True`, we want to return `True`.
  - If `test` is `False`, we want to return `False`.

Solution w/ "Boolean Zen"

- Observation: The `if/else` is unnecessary.
  - The variable `test` stores a `bool` value;
    its value is exactly what you want to return. So return that!

```
def both_odd(n1, n2):
    test = (n1 % 2 != 0 and n2 % 2 != 0)
    return test
```

- An even shorter version:
  - We don't even need the variable `test`.
    We can just perform the test and return its result in one step.

```
def both_odd(n1, n2):
    return (n1 % 2 != 0 and n2 % 2 != 0)
```

"Boolean Zen" template

- Replace

```
def name(parameters):
    if test:
        return True
    else:
        return False
```

- with

```
def name(parameters):
    return test
```

Improve the `is_prime` function

- How can we fix this code?

```
def is_prime(n):
    factors = 0;
    for i in range(1, n + 1):
        if n % i == 0:
            factors += 1

    if factors != 2:
        return False
    else:
        return True
```

De Morgan's Law

- **De Morgan's Law**: Rules used to negate boolean tests.
  - Useful when you want the opposite of an existing test.

Original Expression	Negated Expression	Alternative
`a and b`	`not a or not b`	`not(a and b)`
`a or b`	`not a and not b`	`not(a or b)`

  - Example:

Original Code	Negated Code
`if x == 7 and y > 3:` `    ...`	`if x != 7 or y <= 3:` `    ...`

Boolean practice questions

- Write a function named `is_vowel` that returns whether a `str` is a vowel (a, e, i, o, or u), case-insensitively.
  - `is_vowel("q")` returns `False`
  - `is_vowel("A")` returns `True`
  - `is_vowel("e")` returns `True`

- Change the above function into an `is_non_vowel` that returns whether a `str` is any character except a vowel.
  - `is_non_vowel("q")` returns `True`
  - `is_non_vowel("A")` returns `False`
  - `is_non_vowel("e")` returns `False`

Boolean practice answers

```
# Enlightened version.  I have seen the true way (and false way)
def is_vowel(s):
    return s == 'a' or s == 'A' or s == 'e' or s == 'E' or s =='i' or s == 'I'
           or s == 'o' or s == 'O' or s == 'u' or s =='U'


# Enlightened "Boolean Zen" version
def is_non_vowel(s):
    return not(s == 'a') and not(s == 'A') and not(s == 'e') and not(s == 'E')
           and not(s =='i') and not(s == 'I') and not(s == 'o') and
           not(s == 'O') and not(s == 'u') and not(s =='U')

    # or, return not is_vowel(s)
```

Strings

- **string**: a type that stores a sequence of text characters.

```
name = "text"
name = expression
```

  - Examples:

```
name = "Daffy Duck"
x = 3
y = 5
point = "(" + str(x) + ", " + str(y) + ")"
```

Indexes

- Characters of a string are numbered with 0-based *indexes*:

```
name = "Ultimate"
```

index	0	1	2	3	4	5	6	7
	-8	-7	-6	-5	-4	-3	-2	-1
character	U	l	t	i	m	a	t	e

  - First character's index : 0
  - Last character's index : 1 less than the string's length

Accessing characters

- You can access a character with **string**`[`**index**`]`:

```
name = "Merlin"
print(name[0])
```

Output: `M`

Accessing substrings

- Syntax:

```
part = string[start:stop]
```

- Example:

```
s = "Merlin"
mid = [1:3]     # er
```

- If you want to start at the beginning you can leave off start

```
mid = [:3]      # Mer
```

- If you want to start at the end you can leave off the stop

```
mid = [1:]      # erlin
```

`String` methods

Method name	Description
`find(`**`str`**`)`	index where the start of the given string appears in this string (-1 if not found)
`substring(`**`index1`**`, `**`index2`**`)` or `substring(`**`index1`**`)`	the characters in this string from *index1* (inclusive) to *index2* (exclusive); if *index2* is omitted, grabs till end of string
`lower()`	a new string with all lowercase letters
`upper()`	a new string with all uppercase letters

- These methods are called using the dot notation below:

```
starz = "Biles & Manuel"
print(starz.lower())    # biles & manuel
```

`String` method examples

```
# index      012345678901
s1 = "Allison Obourn"
s2 = "Merlin The Cat"

print(s1.find("o"))       # 5
print(s2.lower())         # "merlin the cat"
```

- Given the following string:

```
# index  01234567890123456789012345
book =  "Building Python Programs"
```

  - How would you extract the word `"Python"` ?

Modifying strings

- String operations and functions like `lowercase` build and return a new string, rather than modifying the current string.

```
s = "Aceyalone"
s.upper()
print(s)    # Aceyalone
```

- To modify a variable's value, you must reassign it:

```
s = "Aceyalone"
s = s.upper()
print(s)    # ACEYALONE
```

ALLISON
LLISON
LISON
ISON
SON
ON
N
A
AL
ALL
ALLI
ALLIS
ALLISO
ALLISON
OBOURN
BOURN
OURN
URN
RN
N
O
OB
OBO
OBOU
OBOUR
OBOURN

Name border

- Prompt the user for full name
- Draw out the pattern to the left
- This should be resizable. Size 1 is shown and size 2 would have the first name twice followed by last name twice

Other `String` operations - length

- Syntax:

```
length = len(string)
```

- Example:

```
s = "Merlin"
count = len(s)     # 6
```

Looping through a string

- The `for` loop through a string using range:

```
major = "CSc"
for letter in range(0, len(major)):
    print(major[letter])
```

- You can also use a `for` loop to print or examine each character without range.

```
major = "CSc"
for letter in major:
    print(letter)
```

Output:

```
C
S
c
```

`String` tests

Method	Description
`startswith(`**`str`**`)`	whether one contains other's characters at start
`endswith(`**`str`**`)`	whether one contains other's characters at end

```
name = "Voldermort"
if name.startswith("Vol"):
    print("He who must not be named")
```

- The `in` keyword can be used to test if a string contains another string.

  example: `"er" in name` **`# true`**

`String` question

- A *Caesar cipher* is a simple encryption where a message is encoded by shifting each letter by a given amount.
  - e.g. with a shift of 3, A → D, H → K, X → A, and Z → C

- Write a program that reads a message from the user and performs a Caesar cipher on its letters:

```
Your secret message: Brad thinks Angelina is cute
Your secret key: 3
The encoded message: eudg wklqnv dqjholqd lv fxwh
```

`Strings` and `ints`

- All `char` values are assigned numbers internally by the computer, called *ASCII* values.
  - Examples:
    `'A'` is 65, `'B'` is 66, `' '` is 32
    `'a'` is 97, `'b'` is 98, `'*'` is 42
  - One character long `Strings` and `ints` can be converted to each other
    `ord('a')` is 97, `chr(103)` is 'g'
  - This is useful because you can do the following:
    `chr(ord('a' + 2))` is `'c'`