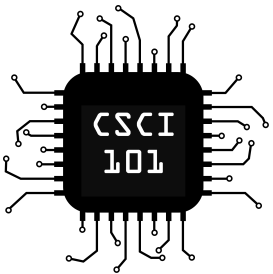
Conditionals



Boolean Data

A **boolean** is a type of data which can be one of two values:
`True` or `False`

Boolean Data

A **boolean** is a type of data which can be one of two values:
`True` or `False`

```
mybool = True
print(mybool)
```

Boolean Data

A **boolean** is a type of data which can be one of two values:
`True` or `False`

```
mybool = True
print(mybool)
```

```
True
```

Another Kind of Equals

`==` is for **equality testing**. An expression `a == b` will be `True` iff `a` has the same value as `b`, `False` otherwise.

Another Kind of Equals

`==` is for **equality testing**. An expression `a == b` will be `True` iff `a` has the same value as `b`, `False` otherwise.

```
>>> year = 2017
>>> year == 2016
False
>>> year == 2017
True
```

Branching

```
if condition:
    # do something
elif other_condition:
    # do something else
else:
    # do something else
```

Branching

Example

```
name = input("What is your name? ")
if name == "Jack":
    print("Your name is the best!")
elif len(name) == 4:
    print("Your name is 4 letters and not 'Jack'!")
else:
    print("Pleased to meet you", name)
```

Comparison Operators

<	Less than	<=	Less or equal	==	Equal
>	Greater than	>=	Greater or equal	!=	Not Equal

Comparison Operators

<	Less than	<=	Less or equal	==	Equal
>	Greater than	>=	Greater or equal	!=	Not Equal

```
age = int(input("What is your age? "))
if age < 0:
    print("I don't think so")
elif age <= 10:
    print("Wow! You're young!")
elif age != 16:
    print("Cool cool.")
else:
    print("Sweet sixteen.")
```

Indentation Denotes Scope

In Python, indentation not only provides style to help yourself and others read your code, but also provides functionality by **denoting the scope of the operation**. Consider the following example:

```
# i was defined previously in this program
if i > 0:
    print("i is positive")
    if i % 2 == 0:
        print("i is even")
    print("hello")
print("goodbye")
```

Indentation Denotes Scope

In Python, indentation not only provides style to help yourself and others read your code, but also provides functionality by **denoting the scope of the operation**. Consider the following example:

```
# i was defined previously in this program
if i > 0:
    print("i is positive")
    if i % 2 == 0:
        print("i is even")
    print("hello")
print("goodbye")
```

1. What will be printed if `i` is `3`?
2. What will be printed if `i` is `-2`?
3. What will be printed if `i` is `4`?

Opposite Day: Using `not`

`not` is an operator which gives the opposite boolean of what it receives. In other words:

- `not False` is `True`
- `not True` is `False`

Opposite Day: Using `not`

`not` is an operator which gives the opposite boolean of what it receives. In other words:

- `not False` is `True`
- `not True` is `False`

So what is `not not not False`?

Try in your Interactive Interpreter!

Opposite Day: Using `not`

`not` is an operator which gives the opposite boolean of what it receives. In other words:

- `not False` is `True`
- `not True` is `False`

So what is `not not not False`?

Try in your Interactive Interpreter!

Example of using `not` in an `if` statement:

```
fish = input("What is your fish's name? ")
if not len(fish) > 3:
    print("What a short name!")
```

Multiple Conditions: Using and and or

What if you want to test the existence of multiple conditions? This is what **and** and **or** are for.

```
fav = int(input("Favorite number? "))
hate = int(input("Least favorite number? "))
if fav * hate == 63 and fav > hate and fav > 0:
    print("Yeah, because 7 ate 9, right?")
elif fav % 2 != 0 or hate % 2 != 0:
    print("What an odd choice.")
else:
    print("Even Steven.")
```

Short Circuiting

`and` and `or` are evaluated left to right, and not all statements will be evaluated if they don't need to.

In other words, if the first part of an `and` is `False`, Python knows the statement is `False` won't bother wasting its time on the second part.

Likewise, if the first part of an `or` is `True`, Python knows the statement is `True` and won't bother wasting its time on the second part.

Computer programmers call this **short-circuiting**.

Practice: Short Circuiting

Which code block is more efficient, given `i` is even half of the time, modulus (%) is very fast, and `hardfunc` takes a few seconds to compute?

```
if hardfunc(i) and i % 2 == 0:
    print("Hello!")
```

```
if i % 2 == 0 and hardfunc(i):
    print("Hello!")
```

Note: `hardfunc` is not built-in to Python, we are just using it as an imaginary example function here.

Practice: Spot the Bug(s)!

What is wrong with the snippet of code below?

```
pets = input("How many pets do you have? ")
if pets < 0:
    print("That's impossible!")
if pets = 0:
    print("Try pets sometime!")
else:
    print("Can I meet them?")
```

Practice: Spot the Bug(s)!

Corrected code snippet:

```
pets = int(input("How many pets do you have? "))
if pets < 0:
    print("That's impossible!")
elif pets == 0:
    print("Try pets sometime!")
else:
    print("Can I meet them?")
```