

Control Flow

Equal Operator ==

The equal operator, `==`, is used to compare two values, variables or expressions to determine if they are the same.

If the values being compared are the same, the operator returns `True`, otherwise it returns `False`.

The operator takes the data type into account when making the comparison, so a string value of `"2"` is *not* considered the same as a numeric value of `2`.

```
# Equal operator
```

```
if 'Yes' == 'Yes':  
    # evaluates to True  
    print('They are equal')  
  
if (2 > 1) == (5 < 10):  
    # evaluates to True  
    print('Both expressions give the  
    same result')  
  
c = '2'  
d = 2  
  
if c == d:  
    print('They are equal')  
else:  
    print('They are not equal')
```

Not Equals Operator !=

The Python not equals operator, `!=`, is used to compare two values, variables or expressions to determine if they are NOT the same. If they are NOT the same, the operator returns `True`. If they are the same, then it returns `False`.

The operator takes the data type into account when making the comparison so a value of `10` would NOT be equal to the string value `"10"` and the operator would return `True`. If expressions are used, then they are evaluated to a value of `True` or `False` before the comparison is made by the operator.

```
# Not Equals Operator
```

```
if "Yes" != "No":  
    # evaluates to True  
    print("They are NOT equal")  
  
val1 = 10  
val2 = 20  
  
if val1 != val2:  
    print("They are NOT equal")  
  
if (10 > 1) != (10 > 1000):  
    # True != False  
    print("They are NOT equal")
```

Comparison Operators

In Python, *relational operators* compare two values or expressions. The most common ones are:

- `<` less than
- `>` greater than
- `<=` less than or equal to
- `>=` greater than or equal too

If the relation is sound, then the entire expression will evaluate to `True`. If not, the expression evaluates to `False`.

and Operator

The Python `and` operator performs a Boolean comparison between two Boolean values, variables, or expressions. If both sides of the operator evaluate to `True` then the `and` operator returns `True`. If either side (or both sides) evaluates to `False`, then the `and` operator returns `False`. A non-Boolean value (or variable that stores a value) will always evaluate to `True` when used with the `and` operator.

```
a = 2
b = 3
a < b # evaluates to True
a > b # evaluates to False
a >= b # evaluates to False
a <= b # evaluates to True
a <= a # evaluates to True
```

```
True and True # Evaluates to True
True and False # Evaluates to
False
False and False # Evaluates to
False
1 == 1 and 1 < 2 # Evaluates to True
1 < 2 and 3 < 1 # Evaluates to
False
"Yes" and 100 # Evaluates to True
```

or Operator

The Python `or` operator combines two Boolean expressions and evaluates to `True` if at least one of the expressions returns `True`. Otherwise, if both expressions are `False`, then the entire expression evaluates to `False`.

```
True or True # Evaluates to True
True or False # Evaluates to True
False or False # Evaluates to
False
1 < 2 or 3 < 1 # Evaluates to True
3 < 1 or 1 > 6 # Evaluates to
False
1 == 1 or 1 < 2 # Evaluates to True
```

not Operator

The Python Boolean `not` operator is used in a Boolean expression in order to evaluate the expression to its inverse value. If the original expression was `True`, including the `not` operator would make the expression `False`, and vice versa.

```
not True # Evaluates to False
not False # Evaluates to True
1 > 2 # Evaluates to False
not 1 > 2 # Evaluates to True
1 == 1 # Evaluates to True
not 1 == 1 # Evaluates to False
```

if Statement

The Python `if` statement is used to determine the execution of code based on the evaluation of a Boolean expression.

- If the `if` statement expression evaluates to `True`, then the indented code following the statement is executed.
- If the expression evaluates to `False` then the indented code following the `if` statement is skipped and the program executes the next line of code which is indented at the same level as the `if` statement.

else Statement

The Python `else` statement provides alternate code to execute if the expression in an `if` statement evaluates to `False`.

The indented code for the `if` statement is executed if the expression evaluates to `True`. The indented code immediately following the `else` is executed only if the expression evaluates to `False`. To mark the end of the `else` block, the code must be unindented to the same level as the starting `if` line.

```
# if Statement
```

```
test_value = 100
```

```
if test_value > 1:
```

```
    # Expression evaluates to True
```

```
    print("This code is executed!")
```

```
if test_value > 1000:
```

```
    # Expression evaluates to False
```

```
    print("This code is NOT executed!")
```

```
print("Program continues at this  
point.")
```

```
# else Statement
```

```
test_value = 50
```

```
if test_value < 1:
```

```
    print("Value is < 1")
```

```
else:
```

```
    print("Value is >= 1")
```

```
test_string = "VALID"
```

```
if test_string == "NOT_VALID":
```

```
    print("String equals NOT_VALID")
```

```
else:
```

```
    print("String equals something  
else!")
```

Boolean Values

Booleans are a data type in Python, much like integers, floats, and strings. However, booleans only have two values:

- `True`
- `False`

Specifically, these two values are of the `bool` type. Since booleans are a data type, creating a variable that holds a boolean value is the same as with other data types.

elif Statement

The Python `elif` statement allows for continued checks to be performed after an initial `if` statement. An `elif` statement differs from the `else` statement because another expression is provided to be checked, just as with the initial `if` statement.

If the expression is `True`, the indented code following the `elif` is executed. If the expression evaluates to `False`, the code can continue to an optional `else` statement. Multiple `elif` statements can be used following an initial `if` to perform a series of checks. Once an `elif` expression evaluates to `True`, no further `elif` statements are executed.

Handling Exceptions in Python

A `try` and `except` block can be used to handle error in code block. Code which may raise an error can be written in the `try` block. During execution, if that code block raises an error, the rest of the `try` block will cease executing and the `except` code block will execute.

```
is_true = True
is_false = False

print(type(is_true))
# will output: <class 'bool'>
```

```
# elif Statement

pet_type = "fish"

if pet_type == "dog":
    print("You have a dog.")
elif pet_type == "cat":
    print("You have a cat.")
elif pet_type == "fish":
    # this is performed
    print("You have a fish")
else:
    print("Not sure!")
```

```
def check_leap_year(year):
    is_leap_year = False
    if year % 4 == 0:
        is_leap_year = True

try:
    check_leap_year(2018)
    print(is_leap_year)
    # The variable is_leap_year is
    # declared inside the function
except:
    print('Your code raised an error!')
```