Logical and Relational Operators

Logical operators are useful when writing code. Logical operators, when combined with flow control, allow for complex choices to be compactly expressed.

Relational Operators

The core logical operators are

Symbol	Function	Definition
$>$	`greater`	Greater than
$>=$	`greater_equal`	Greater than or equal to
$<$	`less`	Less than
$<=$	`less_equal`	Less than or equal to
$==$	`equal`	Equal to
$!=$	`not_equal`	Not equal to

```
>>> x = array([[1,2],[3,
4]])
>>> x > 0
array([[ True, True],
[False, False]], dtype=bool)
>>> x == 3
array([[False, False],
[ True, False]], dtype=bool)
```

```
>>> y = array([1,1])
>>> x < y # y broadcast to be (2,2)
array([[False, False],
[ True, True]], dtype=bool)
>>> z = array([[1,1],[1,
1]])
# Same as broadcast y
>>> x < z
array([[False, False],
[ True, True]], dtype=bool)
```

Logical Operators

Logical expressions can be combined using four logical devices,

Keyword (Scalar)	Function	Bitwise	True if . . .
and	logical_and	Both	True
or	logical_or		Either or Both True
not	logical_not		Not True
	logical_xor	$\wedge$	One True and One False

There are three versions of all operators except XOR. The keyword version (e.g. and) can only be used with scalars and so it not useful when working with NumPy. Both the function and bitwise operators can be used with NumPy arrays, although care is requires when using the bitwise operators.

```
>>> x = arange(2.0,4)
>>> y = x >= 0
>>> z = x < 2
>>> logical_and(y, z)
array([False, False, True, True, False, False], dtype=boo
>>> y & z
array([False, False, True, True, False, False], dtype=boo
>>> (x > 0) & (x < 2)
array([False, False, True, True, False, False], dtype=boo
>>>
```

```
>>> x > 0 & x < 4 # Error
TypeError: ufunc bitwise_and not supported for the input
and the inputs could not
be safely coerced to any supported types
according to the casting rule safe

>>> ~(y & z) # Not
array([ True, True, False, False, True, True], dtype=bool
>>>
```

Bitwise operators

Bitwise operators have high priority higher than logical comparisons and so parentheses are requires around comparisons. For example, $(x > 1)\&(x < 5)$ is a valid statement, while $x > 1\&x < 5$, which is evaluated as $(x > (1\&x)) < 5$, produces an error.

Bitwise operators

```
>>> x = arange(2.0,4)
>>> y = x >= 0
>>> z = x < 2
>>> logical_and(y, z)
array([False, False, True, True, False, False], dtype=boo
>>> y & z
array([False, False, True, True, False, False], dtype=boo
>>> (x > 0) & (x < 2)
array([False, False, True, True, False, False], dtype=boo
```

Multiple tests: `all` and `any`

The commands all and any take logical input and are self-descriptive. all returns True if all logical elements in an array are 1.

- If `all` is called without any additional arguments on an array, it returns True if all elements of the array are logical true and 0 otherwise.
- any returns logical(True) if any element of an array is True.

Multiple tests: `all` and `any`

- Both `all` and `any` can be also be used along a specific dimension using a second argument or the keyword argument axis to indicate the axis of operation (0 is column-wise and 1 is row-wise).
- When used column- or row-wise, the output is an array with one less dimension than the input, where each element of the output contains the truth value of the operation on a column or row.

Multiple tests: `all` and `any`

```
>>> x = array([[1,2][3,4]])
>>> y = x <= 2
>>> y
array([[ True, True],
[False, False]], dtype=bool)
>>> any(y)
True
>>> any(y,0)
array([[ True, True]], dtype=bool)
>>> any(y,1)
array([[ True],
[False]], dtype=bool)
```

allclose

- `allclose` can be used to compare two arrays for near equality.
- This type of function is important when comparing floating point values which may be effectively the same although not identical.

```
>>> eps = np.finfo(np.float64).eps
>>> eps
2.2204460492503131e16
>>> x = randn(2)
>>> y = x + eps
115
>>> x == y
array([False, False], dtype=bool)
>>> allclose(x,y)
True
```

The tolerance for being close can be set using keyword arguments either relatively (rtol) or absolutely (atol).

array_equal

`array_equal` tests if two arrays have the same shape and elements. It is safer than comparing arrays directly since comparing arrays which are not broadcastable produces an error.

array_array_equiv

`array_equiv` tests if two arrays are equivalent, even if they do not have the exact same shape. Equivalence is defined as one array being broadcastable to produce the other.

```
>>> x = randn(10,1)
>>> y = tile(x,2)
>>> array_equal(x,y)
False
>>> array_equiv(x,y)
True
```

is*

- A number of special purpose logical tests are provided to determine if an array has special characteristics.
- Some operate element-by-element and produce an array of the same dimension as the input while other produce only scalars.
- These functions all begin with `is`.

Operator	True if . . .	Method of operat
`isnan`	1 if nan	element-by-eleme
`isinf`	1 if inf	element-by-eleme
`isfinite`	1 if not inf and not nan	element-by-eleme
`isposfin,isnegfin`	1 for positive or negative inf	element-by-eleme
`isreal`	1 if not complex valued	element-by-eleme

iscomplex	1 if complex valued	element-by-element
isreal	1 if real valued	element-by-element
is_string_like	1 if argument is a string	scalar
is_numlike	1 if is a numeric type	scalar
isscalar	1 if scalar	scalar
isvector	1 if input is a vector	scalar