Digital Career Institute

Python Course - Logical Thinking

Digital Career Institute

Goal of the Submodule

The goal of this submodule is to help the student understand how to work with complex logical expressions. By the end of this submodule, the learners will be able to understand:

- What are the different parts of a logical expression.
- How the interpreter reads an expression and evaluates it.
- Which are the advanced operators and how to implement them in Python.
- How to read and use complex logical evaluations.
- What a truth table is and how to use it.
- What short-circuiting is and how it is used.
- That non-Boolean values can be considered “falsy” and “truthy” and may evaluate to `True` or `False` in a logical expression.

Digital Career Institute

Topics

- Anatomy of a logical expression. Predicates and operators.
- Reminder of basic logical operators.
- Evaluation precedence. Short-circuiting and operator precedence.
- Truth tables.
- Advanced logical operators, their general name and their implementation in Python.
- Strategies to create readable complex logical expressions.

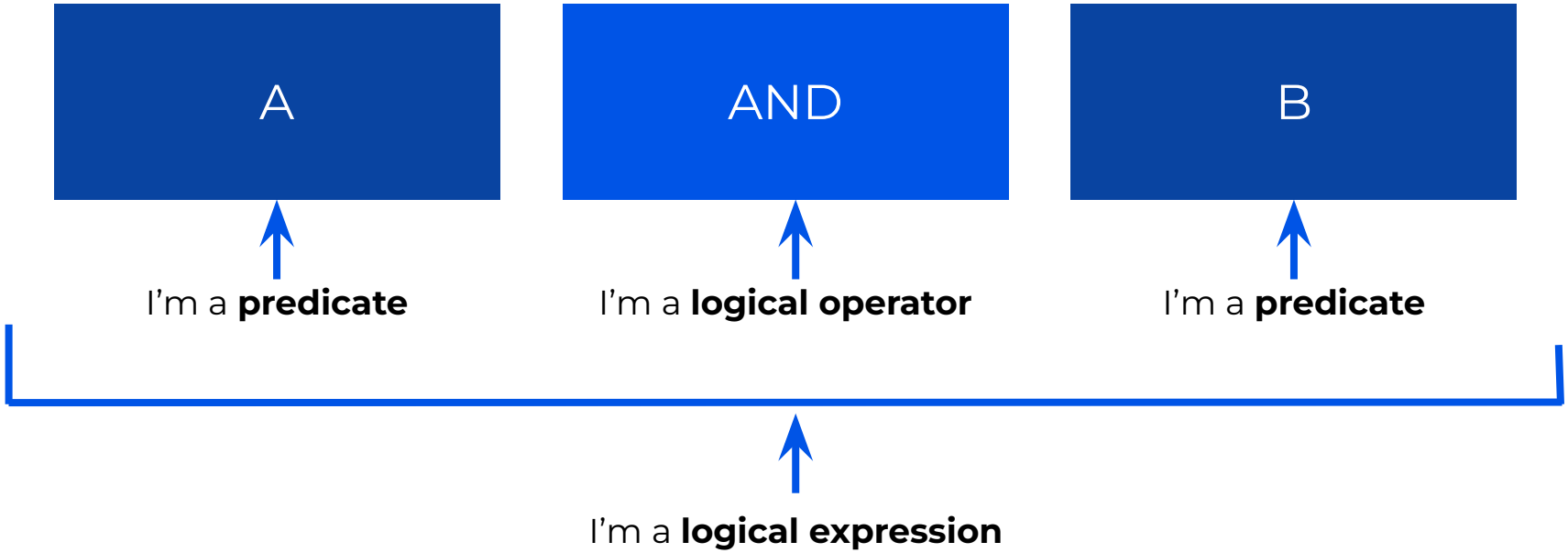
Logical Thinking

Glossary

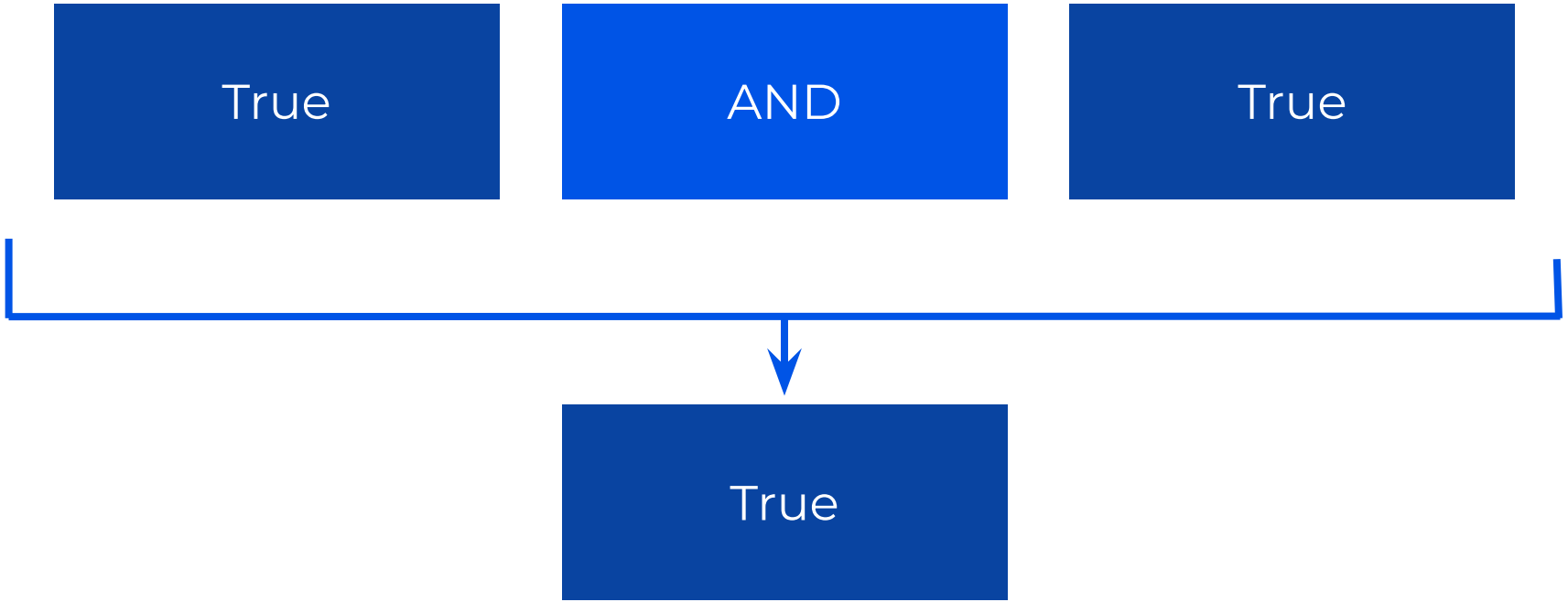
Digital Career Institute

Term	Definition
Boolean	A data type that has two possible values: `True` / `False`.
Predicate	An expression that evaluates/returns a Boolean.
Logical operator	A symbol that performs a logical operation between the predicates on either side of it (binary operators) or to its right (unary operators).
Logical expression	A set of predicates and operators that produce a single Boolean.
Evaluation	The result of executing an expression or a predicate.

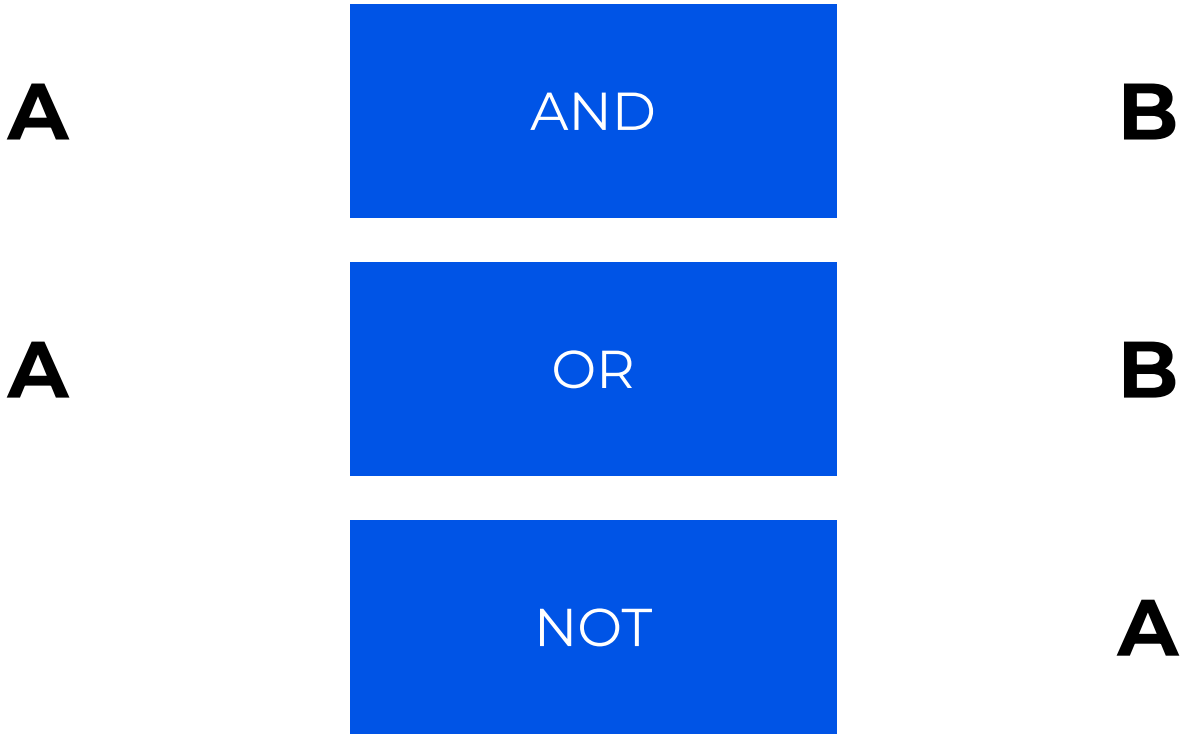
Defining a Logical Expression



Evaluating a Logical Expression



Python Logical Operators



Logical Expressions

```
>>> a = True
>>> b = True
>>> if a and b:
...     print("Both conditions are
met")
...
Both conditions are met
```

Logical expressions are often used in conditional statements.

Logical Expressions

```
>>> import random
>>> a = 0
>>> b = 0
>>> while a < 0.9 and b < 10:
...     b = b + 1
...     a = random.random()
...
>>> print(b)
8
```

But can also be used in many other situations.

The predicate `a < 0.9` will randomly evaluate to `True` or `False`.

The predicate `b < 10` will evaluate to `True` the first 10 iterations. Then it will evaluate to `False`.

The complete logical expression will evaluate to `False` in 10 or less iterations.

Predicates

Predicates

```
>>> a = True
>>> b = 0
>>> while a and b < 10 and isOdd(b):
...     b = b + 1
...
>>> print(b)
1
```

Anything that evaluates to True or False can be a predicate.

- `True` is a predicate.
- `b < 10` is a predicate.
- `isOdd(2)` is a predicate.

Many operators and predicates can be added in the same expression.

Predicates

```
>>> import random
>>> a = 0
>>> b = 0
>>> while a < 0.9 and (
...                 b < 10 or isEven(b)
...             ):
...         b = b + 1
...         a = random.random()
...
>>> print(b)
9
```

A predicate can also be another logical expression.

`b < 10 or isEven(b)` is a logical expression that is used as a predicate on the main logical expression in this evaluation.

Predicates

```
>>> a = 0
>>> b = ""
>>> c = []
>>> d = {}
>>> if a or b or c or d:
...     print("We have something")
... else:
...     print("Empty variables")
...
Empty variables
```

Non-Boolean *values may also be used as predicates.*

Empty values are considered to be "falsy" and they are evaluated to `False`.

Predicates

```
>>> a = 1
>>> b = "Hello"
>>> c = ["A"]
>>> d = {"b": 3}
>>> if a and b and c and d:
...     print("None empty")
... else:
...     print("Some empty")
...
None empty
```

Non-Boolean *values may also be used as predicates.*

Non-empty values are considered to be "truthy" and they are evaluated to `True`.

Predicates

```
>>> a = None
>>> if a:
...     print("Var is set")
... else:
...     print("Var is not set")
...
Var is not set
```

Non-Boolean *values may also be used as predicates.*

`None` always evaluates to `False`.

A ***non-Boolean*** *value can always be forced into a boolean with the built-in function* ***bool()***.

Predicates

```
>>> False or 2
2
>>> True and 2
2
>>> True or 2
True
>>> False and 2
False
>>> 2 and 0
0
```

Non-Boolean *values may also be used as predicates.*

They are treated as `True` or `False`, but their evaluation still returns the original type and value.

If they are short-circuited, the evaluation takes the value of the prevailing predicate.

Predicates

```
>>> user = input("Username: ") or "Unknown"
Username: Mary
>>> print(user)
Mary

>>> user = input("Username: ") or "Unknown"
Username:
>>> print(user)
Unknown
```

*Evaluating a **Non-Boolean** value still returns the original object.*

This feature is often used to provide default values to any evaluation that may return a *falsy* value.

`Unknown` always evaluates as *truthy*. If the user does not type a name, the first operand is *falsy* and the evaluation will return `Unknown`.

Evaluation Precedence

Short-Circuit Evaluation

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...         a < 0.9
...         and b < 10
...         and isEven(b)
...         ):
...     b = b + 1
...     a = random.random()
...
>>> print(b)
35
```

Predicates are evaluated in the order they are written.

If `a < 0.9` evaluates to `False`, the rest of the predicates are irrelevant and Python does not evaluate them.

*This is called **Short-circuit evaluation**.*

Short-Circuit Evaluation

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...         a < 0.9
...         or b < 10
...         or isEven(b)
...         ):
...     b = b + 1
...     a = random.random()
...
>>> print(b)
35
```

Also with the `or` operator.
If `a < 0.9` evaluates to `True`, Python will not check if `b` is lower than `10` or `isEven(b)`.

Short-Circuit Evaluation

```
>>> a = 0
>>> b = 1
>>> (b / a) > 0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in
<module>
     (b / a) > 0
ZeroDivisionError: division by zero
>>> a != 0 and (b / a) > 0
False
```

Short-circuit evaluation may be used to catch edge cases and error scenarios.

In this case, `a != 0` evaluates to `False`, so Python does not even try to evaluate the second predicate, which would produce an error.

Short-Circuit Evaluation

We can also use the short-circuit evaluation to optimize our code.

If there is a **`complex`** expression that takes some time to evaluate.

And there is a **`simple`** expression.

The order of the predicates may have a direct impact on the overall performance of the script.

```
>>> def complex():
...     time.sleep(1)
...     return True
...
>>> @timeit
>>> def test():
...     simple = True
...     for i in range(0, 10):
...         if i >= 5:
...             simple = False
...         if simple or complex():
...             pass
...
```

We use a decorator to time the execution of the function.

We test the OR expression 10 times. 5 with the **`simple`** expression as **`True`** and 5 more as **`False`**.

Short-Circuit Evaluation

Digital Career Institute

Using the short-circuit evaluation to optimize our code.

```
>>> def complex():
...     time.sleep(1)
...     return True
...
>>> @timeit
>>> def test():
...     simple = True
...     for i in range(0, 10):
...         if i >= 5:
...             simple = False
...         if simple or complex():
...             pass
...
>>> test()
It took 5.005 seconds.
```

```
>>> def complex():
...     time.sleep(1)
...     return True
...
>>> @timeit
>>> def test():
...     simple = True
...     for i in range(0, 10):
...         if i >= 5:
...             simple = False
...         if complex() or simple:
...             pass
...
>>> test()
It took 10.002 seconds.
```

Logical Operators Precedence

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...           (a < 0.9
...           and b < 10)
...           or (not isOdd(b))
...           ):
...       b = b + 1
...       a = random.random()
...
>>> print(b)
11
```

Is it `(a and b) or not isOdd`?
or `a and (b or not isOdd)`?

Each operator has a different precedence priority on the evaluation.

1. **not**
2. **and**
3. **or**

Logical Operators Precedence

Digital Career Institute

Equivalent code. This is how they are evaluated.

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...         a < 0.9
...         and b < 10
...         or not isOdd(b)
...         ):
...     b = b + 1
...     a = random.random()
...
>>> print(b)
11
```

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...         (a < 0.9 and b < 10)
...         or
...         (not isOdd(b))
...         ):
...     b = b + 1
...     a = random.random()
...
>>> print(b)
11
```

Logical Operators Precedence

Digital Career Institute

These will produce different results

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...         a < 0.9
...         and b < 10
...         or not isOdd(b)
...         ):
...     b = b + 1
...     a = random.random()
...
>>> print(b)
11
```

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...         a < 0.9
...         and
...         (b < 10 or not isOdd(b))
...         ):
...     b = b + 1
...     a = random.random()
...
>>> print(b)
7
```

Logical Operators Precedence

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...         a < 0.9
...         and
...         (b < 10 or not isOdd(b))
...         ):
...     b = b + 1
...     a = random.random()
...
>>> print(b)
7
```

Parentheses may be used to force Python to change the natural precedence of the logical operators.

We learned ...

- What is a predicate, an operator and an expression.
- That expressions and predicates are evaluated as `True` or `False`.
- That there are 3 basic Python operators: `and`, `or` and `not`.
- That anything that can be evaluated to `True` or `False` can be used as a predicate: expressions, functions, Non-Boolean values,...
- That Python does not evaluate all the expressions if the first conditions are enough to determine the result value.
- That the `not` operator takes the highest priority during the evaluation, followed by `and` and then `or`.

Truth Tables

Truth Tables

Digital Career Institute

Logical expressions may become very complex.

The best way to understand and check how our logical expression works is to write down every possible evaluation of its predicates and the result.

*These are called **truth tables***

Truth Tables

```
>>> import random
>>> a = 0
>>> b = 0
>>> while a < 0.9 and b < 10:
...     b = b + 1
...     a = random.random()
...
>>> print(b)
8
```

Truth Table - AND

`a < 0.9`	`b < 10`	Expression
True	True	True
True	False	False
False	*True**	False
False	*False**	False

** These are short-circuited, not evaluated.*

Truth Tables

```
>>> import random
>>> a = 0
>>> b = 0
>>> while a < 0.9 and b < 10:
...     b = b + 1
...     a = random.random()
...
>>> print(b)
8
```

Evaluation Table - AND

`a < 0.9`	`b < 10`	Expression
True	True	True
True	False	False
False	*	False

We can also use an evaluation table, a byproduct of a truth table that summarizes and incorporates the short-circuiting.

Truth Tables

```
>>> import random
>>> a = 0
>>> b = 0
>>> while a < 0.9 or b < 10:
...     b = b + 1
...     a = random.random()
...
>>> print(b)
8
```

Truth Table - OR

`a < 0.9`	`b < 10`	Expression
True	*True*	True
True	*False*	True
False	True	True
False	False	False

Truth Tables

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...         a < 0.9
...         and b < 10
...         or not isOdd(b)
...         ):
...     b = b + 1
...     a = random.random()
...
>>> print(b)
11
```

Truth Table - AND OR

`a < 0.9`	`b < 10`	`Not isOdd(b)`	Expression
True	True	*True*	*True**
True	True	*False*	*True**
True	False	True	True
True	False	False	False
False	True	True	True
False	True	False	False
False	False	True	True
False	False	False	False

Truth Tables

```
>>> import random
>>> a = 0
>>> b = 0
>>> while (
...         a < 0.9
...         and
...         ( b < 10 or not isOdd(b) )
...         ):
...     b = b + 1
...     a = random.random()
...
>>> print(b)
11
```

Truth Table - OR AND

`a < 0.9`	`b < 10`	`Not isOdd(b)`	Expression
True	True	*True*	True
True	True	*False*	True
True	False	True	True
True	False	False	False
False	*True*	*True*	False
False	*True*	*False*	False
False	*False*	*True*	False
False	*False*	*False*	False

Truth Tables

Truth Tables *help us evaluate if a logical expression is doing what we think it does.*

Advanced Operators

Logical Operators

Digital Career Institute

Python native	Not implemented
AND	XOR
OR	NAND
NOT	NOR
	XNOR

Basic Operators

Digital Career Institute

AND

```
>>> if A and B:
...     # instructions
...
```

A	B	Expression
True	True	True
True	False	False
False	*True*	False
False	*False*	False

Basic Operators

Digital Career Institute

OR

```
>>> if A or B:
...     # instructions
...
```

A	B	Expression
True	*True*	True
True	*False*	True
False	True	True
False	False	False

Basic Operators

NOT

```
>>> if not A:
...     # instructions
...
```

A	Expression
True	False
False	True

Advanced Operators

Digital Career Institute

XOR

```
>>> if A != B:
...     # instructions
...
```

A	B	Expression
True	True	False
True	False	True
False	True	True
False	False	False

Advanced Operators

Digital Career Institute

NAND

```
>>> if not (A and B):
...     # instructions
...
```

A	B	Expression
True	True	False
True	False	True
False	*True*	True
False	*False*	True

Advanced Operators

Digital Career Institute

NOR

```
>>> if not (A or B):
...     # instructions
...
```

A	B	Expression
True	*True*	False
True	*False*	False
False	True	False
False	False	True

Advanced Operators

Digital Career Institute

XNOR

```
>>> if A == B:
...     # instructions
...
```

A	B	Expression
True	True	True
True	False	False
False	True	False
False	False	True

Complex Expressions

Complex Expressions

Digital Career Institute

There are other mechanisms that can be used to better understand logical expressions, debug them and make them more readable.

Naming **&** Grouping

Variable Naming

```
>>> import random
>>> luck = 0
>>> counter = 0
>>> while (
...         luck < 0.9
...         and counter < 10
...         or isEven(counter)
...         ):
...     counter = counter + 1
...     luck = random.random()
...
>>> print(counter)
11
```

*Use **proper and semantic names** for your variables.*

Naming Predicates

```
>>> import random
>>> lucky = False
>>> counter = 0
>>> while (
...         lucky
...         and counter < 10
...         or isEven(counter)
...         ):
...     counter = counter + 1
...     lucky = random.random() < 0.9
...
>>> print(counter)
11
```

We can store the result of the evaluation in a variable with a proper name.

The name **`lucky`** will need to be instantiated as a Boolean and updated manually inside the loop.

Naming Predicates

```
>>> import random
>>> def lucky():
...     return random.random() < 0.9
...
>>> counter = 0
>>> while (
...         lucky()
...         and counter < 10
...         or isEven(counter)
...         ):
...     counter = counter + 1
...
>>> print(counter)
11
```

We can also store the logic of the predicate behind a named function.

We already did this when calling **`isEven`**.

Grouping Predicates

```
>>> import random
>>> def lucky():
...     return random.random() < 0.9
...
>>> def limit_not_reached(counter):
...     return counter < 10 or isEven(counter)
...
>>> counter = 0
>>> while (
...         lucky()
...         and limit_not_reached(counter)
...         ):
...     counter = counter + 1
...
>>> print(counter)
11
```

*We can group various expressions inside a function with **a meaningful name**.*

Grouping Predicates

Digital Career Institute

MAIN - AND		
lucky	**limit_not_reached**	**Expression**
True	True	True
True	False	False
False	True	False
False	False	False

limit_not_reached - OR		
counter < 10	**isEven(counter)**	**Expression**
True	True	True
True	False	True
False	True	True
False	False	False

Grouping helps us split the truth table in more, easier to evaluate, tables.

We learned ...

- That truth tables are a way to display all evaluation scenarios.
- That they may be useful to check if the expression is actually doing what we expect it to do.
- That the more predicates and operators we use the bigger the truth table will get.
- That Python does not implement the advanced logical operators.
- How to implement in Python the advanced logical operators.
- How to define complex logical expressions that are easy to read, understand and debug.

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute