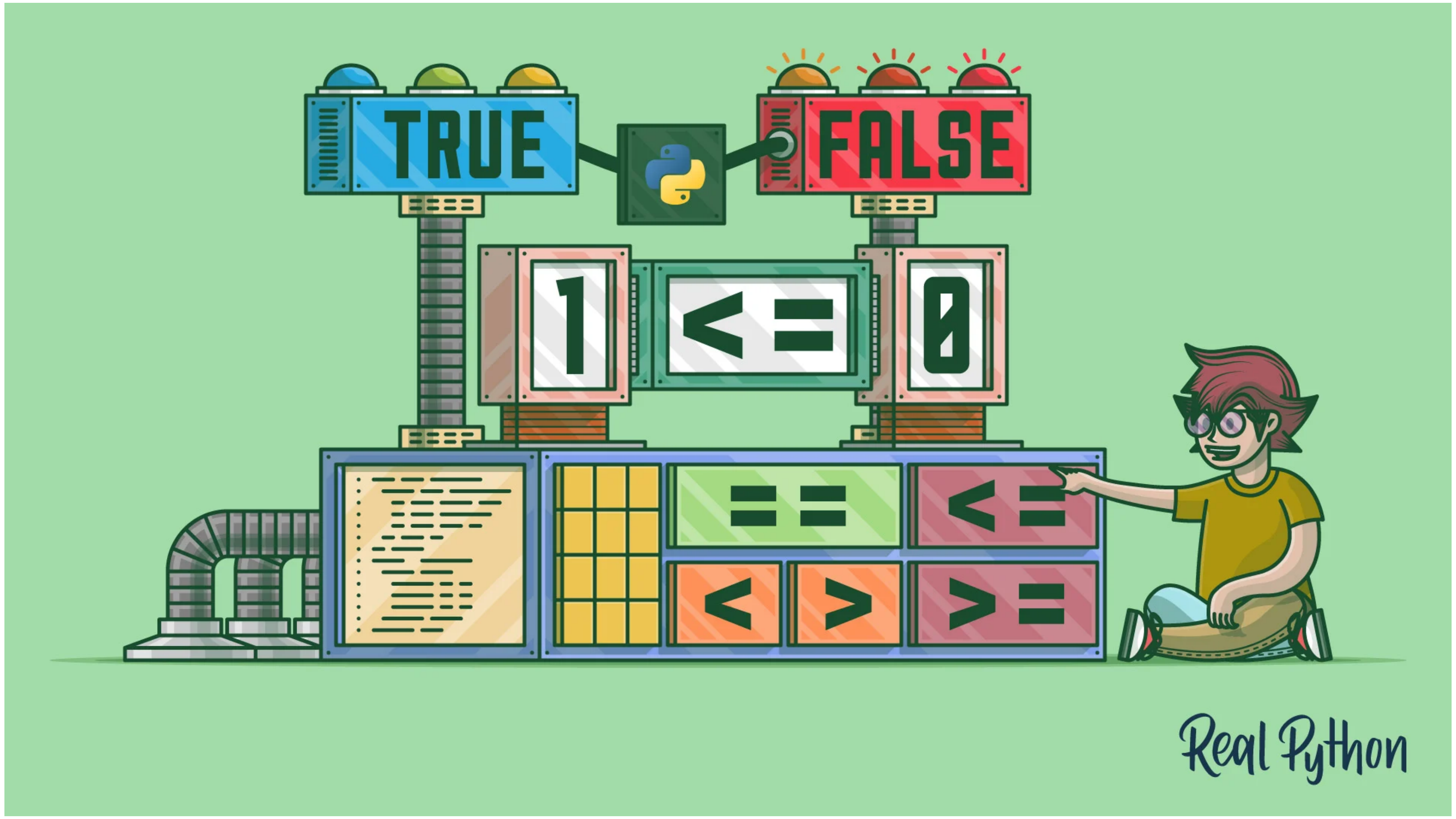
PYTHON BOOLEANS



Are you doing this?

Are you doing this?

```
if len(my_list) > 0:

    do_something()

else:

    do_something_else()
```

Are you doing this?

Are you doing this?

```
if first_name != "":

    print(f"Hello {first_name}!")

else:

    print(f"Hello Anonymous!")
```

Are you doing this?

Are you doing this?

```
if func_1() < func_2() and func_2() <= MAX_LENGTH:

    do_something()

else:

    do_something_else()
```

Which of the following is valid Python syntax?

Which of the following is valid Python syntax?

```
>>> 1 < 2 in [1, 2] not in [[1], [3, 4]]
```

Which of the following is valid Python syntax?

```
>>> 1 < 2 in [1, 2] not in [[1], [3, 4]]
```

```
>>> True + True + False/True + True
```

Which of the following is valid Python syntax?

```
>>> 1 < 2 in [1, 2] not in [[1], [3, 4]]
```

```
>>> True + True + False/True + True
```

```
>>> birthday = get_birthday() or "Unknown"
```

Succinct Conditional Statements

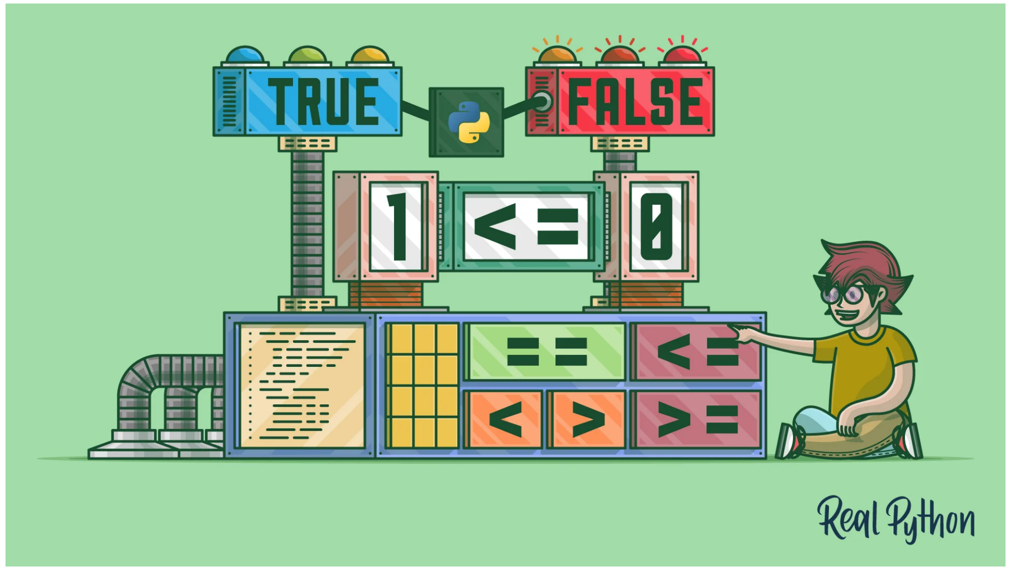
```
if my_list:

    do_something()

else:

    do_something_else()
```

Leverage Python’s Comparison Operators

```
if func_1() < func_2() <= MAX_LENGTH:
    do_something()
else:
    do_something_else()
```

What will you learn?

1. The Python Boolean Type
2. Boolean Operator `not`
3. Boolean Operator `and`
4. Boolean Operator `or`
5. Comparison Operators
6. Chaining Comparison Operators
7. Python Boolean Testing
8. Summary



PYTHON BOOLEANS

The Python Boolean Type

- In 2002, the `bool` class was added as a Python built-in data type (**PEP 285**)
- Why? To standardize the process of deciding when a condition is **true** or **false**
- The `bool` class is a **subclass** of the `int` class
- The function `bool()` can be used to transform an object to a Boolean
- Provided the input object can be assigned a truth value

The Python Boolean Type

- The value of an instance of `bool` is either `True` or `False`
- `True` and `False` are keywords
- As `int` objects, `True` evaluates to **1** and `False` to **0**

The Python Boolean Type

- The value of an instance of `bool` is either `True` or `False`
- `True` and `False` are keywords
- As `int` objects, `True` evaluates to **1** and `False` to **0**

```
>>> True + True + False + True
3
```

Boolean Operators

- A **boolean operator** takes as input one or more booleans and returns a boolean
- It is convenient to completely specify a boolean operator using a **truth table**
- In the theory of boolean algebra, the **NOT**, **AND**, and **OR** operators play an important role
- These are implemented in Python as `not`, `and`, and `or`
- As you will see, Python's implementation of `and` and `or` makes it easy to write efficient and readable code

`not` Operator

- The `not` operator is the only **unary** Boolean operator implemented in Python
- In Python: `not x`
- `not` can be applied to any object
- `not` returns only `True` or `False`

`not` Operator

- The `not` operator is the only **unary** Boolean operator implemented in Python
- In Python: `not x`
- `not` can be applied to any object
- `not` returns only `True` or `False`

Truth table

`x`	`not x`
T	F
F	T

`and` Operator

- The `and` operator is one of only two **binary** Boolean operators in Python
- In Python: `x and y`
- `and` returns `True` only when **both** operands evalute to `True`

`and` Operator

- The `and` operator is one of only two **binary** Boolean operators in Python
- In Python: `x and y`
- `and` returns `True` only when **both** operands evalute to `True`

Truth table

`(x,y)`	`x and y`
(T, T)	T
(T, F)	F
(F, T)	F
(F, F)	F

`and` Operator

- The `and` operator is one of only two **binary** Boolean operators in Python
- In Python: `x and y`
- `and` returns `True` only when **both** operands evalute to `True`

Truth table

`(x,y)`	`x and y`
(T, T)	T
(T, F)	F
(F, T)	F
(F, F)	F

`and` Operator

- The `and` operator is one of only two **binary** Boolean operators in Python
- In Python: `x and y`
- `and` returns `True` only when **both** operands evalute to `True`

Truth table

`(x,y)`	`x and y`
(T, T)	T
(T, F)	F
(F, T)	F
(F, F)	F

`and` Operator

- Although Boolean operators are by definition supposed to return Boolean values, Python's `and` operator returns the **value** of one of its operands
- The return value is determined by `and`'s truth table: `x and y`
    - First `x` is evaluated
    - If `x` is `False` (or **falsy**) then the value of `x` is returned
    - Otherwise, `y` is evaluated
    - And the resulting value of `y` is returned
- This **short-circuiting** can be used to shorten code or set a default value

`or` Operator

- The `or` operator is the other **binary** Boolean operator in Python
- In Python: `x or y`
- `or` returns `False` only when **both** operands evalute to `False`

`or` Operator

- The `or` operator is the other **binary** Boolean operator in Python
- In Python: `x or y`
- `or` returns `False` only when **both** operands evalute to `False`

Truth table

`(x,y)`	`x or y`
(T, T)	T
(T, F)	T
(F, T)	T
(F, F)	F

`or` Operator

- The `or` operator is the other **binary** Boolean operator in Python
- In Python: `x or y`
- `or` returns `False` only when **both** operands evalute to `False`

Truth table

`(x,y)`	`x or y`
(T, T)	T
(T, F)	T
(F, T)	T
(F, F)	F

`or` Operator

- The `or` operator is the other **binary** Boolean operator in Python
- In Python: `x or y`
- `or` returns `False` only when **both** operands evalute to `False`

Truth table

`(x,y)`	`x or y`
(T, T)	T
(T, F)	T
(F, T)	T
(F, F)	F

`or` Operator

- Just like `and`, `or` will return the value of the operand consistent with `or`'s truth table
- `x or y`:
    - First `x` is evaluated
    - If `x` is `True` (or **truthy**) then the value of `x` is returned
    - Otherwise, `y` is evaluated
    - And the resulting value of `y` is returned

Comparison Operators

- A **comparison operator** is a binary operator that determines whether a particular relationship holds between the operands
- For all **built-in** Python objects, comparison operators return either `True` or `False`
- However, Python allows you to create objects for which comparison operators return **composite** Boolean-type objects (e.g., NumPy)
- There are eight value comparison operators and two membership operators

Comparison Operators

Operator	Testing if ...
`x < y`	`x` is strictly less than `y`
`x <= y`	`x` is less than or equal `y`
`x > y`	`x` is strictly greater `y`
`x >= y`	`x` is greater than or equal `y`
`x == y`	`x` and `y` have equal value
`x != y`	`x` and `y` have unequal values
`x is y`	`x` and `y` are equal objects
`x is not y`	`x` and `y` are unequal objects

Comparison Operators

Operator	Testing if ...
`x < y`	`x` is strictly less than `y`
`x <= y`	`x` is less than or equal `y`
`x > y`	`x` is strictly greater `y`
`x >= y`	`x` is greater than or equal `y`
`x == y`	`x` and `y` have equal value
`x != y`	`x` and `y` have unequal values
`x is y`	`x` and `y` are equal objects
`x is not y`	`x` and `y` are unequal objects

Comparison Operators

Operator	Testing if ...
`x < y`	`x` is strictly less than `y`
`x <= y`	`x` is less than or equal `y`
`x > y`	`x` is strictly greater `y`
`x >= y`	`x` is greater than or equal `y`
`x == y`	`x` and `y` have equal value
`x != y`	`x` and `y` have unequal values
`x is y`	`x` and `y` are equal objects
`x is not y`	`x` and `y` are unequal objects

Comparison Operators

- The equality operators `==`, `!=`, `is`, and `is not` can be applied to different data-types
- However, distinct data-types will never compare equal, except possibly numeric types
- In general, the order operators `<`, `<=`, `>`, and `>=` cannot be applied to different data-types
- Moreover, some objects do not have a well-defined or canonical ordering, for example, **dictionaries**

Membership Operators

Operator	Testing if ...
`x in a`	`x` is a member of `a`
`x not in a`	`x` is not a member of `a`

- Lists, tuples, sets, and dictionaries support the membership operators
- For a dictionary, membership is checked on the **keys**
- For strings, membership is checked for the existence of a substring
- User-defined classes that define `__contains__()` or `__iter__()` support the membership operators

Chaining Comparison Operators

Chaining Comparison Operators

- Consider the expression:

```
if x < y and y <= z:
  do_something()
```

Chaining Comparison Operators

- Consider the expression:

```
if x < y and y <= z:
  do_something()
```

- This can be shortened to:

```
if x < y <= z:
  do_something()
```

Chaining Comparison Operators

- Or

```
if x != n and n in some_list:
  do_something()
```

Chaining Comparison Operators

- Or

```
if x != n and n in some_list:
  do_something()
```

- Can be shortened to:

```
if x != n in some_list
  do_something()
```

Chaining Comparison Operators

- Readability **quickly** degrades after 3 or more comparisons

```
>>> 1 < 2 in [1, 2, 3] >= [1, 2] != 4 == 8/2
    True
```

- According to the Zen of Python (PEP 20), “**Readability counts**”

Chaining Comparison Operators

- Comparison chains are just chains of `and` calls

```
>>> 1 < 2 in [1, 2, 3] >= [1, 2]
    True
>>> (1 < 2) and (2 in [1, 2, 3]) and ([1, 2, 3] >= [1, 2])
    True
```

Chaining Comparison Operators

- Comparison chains are just chains of `and` calls

```
>>> 1 < 2 in [1, 2, 3] >= [1, 2]
    True
>>> (1 < 2) and (2 in [1, 2, 3]) and ([1, 2, 3] >= [1, 2])
    True
```

- However, in chained comparisons, expressions are evaluated only once

```
if purchase_price() <  account_balance() < sell_price():
    make_purchase()
```

Python Boolean Testing

- Booleans are most frequently used in `if` statements:

```
>>> if some_expression:
        do_something()
```

- `some_expression` is evaluated and determined to be either **truthy** which evaluates to `True` or **falsy** which evaluates to `False`
- **Any** object is considered truthy **unless** its class defines either a `__bool__` method returning `False` or a `__len__` method that returns zero
- If both are defined then `__bool__` takes precedence

Some Falsy Built-In Objects

- Instances of built-in objects considered `False` are:
    - `None`, `False`
    - Zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
    - Empty sequences and collections: `""`, `()`, `[]`, `set()`, `dict()`, `range(0)`

```
>>> bool(None)
    False
>>> bool(0.0)
    False
>>> bool([])
    False
```

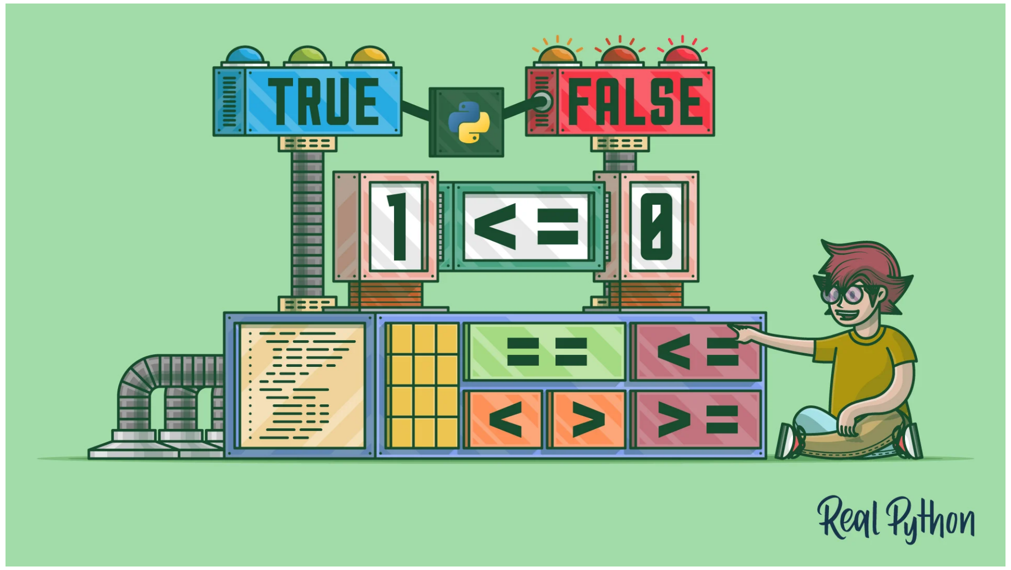
Leveraging Short-Circuiting

- Since `None` is falsy and `or` returns the second operand if the first evaluates to `False`, we can set default values when none is given:

```
>>> def message(msg=None):
        return msg or "This action cannot be undone!"

>>> print(message())
This action cannot be undone!

>>> print(message("Are you sure?"))
Are you sure?
```

Summary

- How the `bool` data-type behaves and its connection to other built-in data-types
- How to use Boolean and comparison operators to write **efficient** Pythonic code
- How to use Python's implicit conversion of objects to Booleans to write **readable** code
- There really is more to Booleans than just `True` and `False`



PYTHON BOOLEANS