A First Example

Programming often involves examining a set of conditions and deciding which action to take based on those conditions. Python's `if` statement allows you to examine the current state of a program and respond appropriately to that state.

In [ ]:

```
titles = ['moby-dick; or, the whale', 'dracula', 'adventures of huckleberry f
          'the adventures of sherlock holmes', "alice's adventures in wonderla

for title in titles:
    if 'moby-dick' in title:
        print(title.upper())
    else:
        print(title.title())
```

Conditional Tests

At the heart of every `if` statement is an **expression** that can be evaluated as `True` or `False` that is called a conditional test. Python uses the values `True` and `False` to decide whether the code in an `if` statement should be executed. If a conditional test evaluates to `True` , Python executes the code following the `if` statement. If the test evaluates to `False` , Python ignores the code following the if statement.

That is, conditional test are just **Boolean Expressions**.

> "A Boolean expression is an expression in a programming language that produces a Boolean value when evaluated, i.e. one of true or false. A Boolean expression may be composed of a combination of the Boolean constants true or false, Boolean-typed variables, Boolean-valued operators, and Boolean-valued functions."
>
> Gries, David; Schneider, Fred B. (1993), "Chapter 2. Boolean Expressions", *A Logical Approach to Discrete Math*, Monographs in Computer Science, Springer, p. 25

In [3]:

```
name = 'Ishmael'

# Check for equality
if name == 'Ishmael':
    print('Reading Moby Dick.')

# Check for inequality
if name != 'Ishmael':
    print('Reading something else.')
```

```
Reading Moby Dick.
```

Comparison Operators

Comparison operators compare two values and evaluate down to a single Boolean value.

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

In [4]:

```
42 == 42
```

Out[4]:

```
True
```

In [5]:

```
42 == 99
```

Out[5]:

```
False
```

In [6]:

```
2 != 3
```

Out[6]:

```
True
```

In [7]:

```
2 != 2
```

Out[7]:

False

In [8]:

```
'Hej' == 'Hej'
```

Out[8]:

True

In [9]:

```
'Hej' == 'hej'
```

Out[9]:

False

In [10]:

```
True == True
```

Out[10]:

True

In [11]:

```
True != False
```

Out[11]:

True

In [14]:

```
42 == 42.0
```

Out[14]:

True

In [17]:

```
42 == '42'
```

Out[17]:

False

Boolean operators

The and and or operators always take two Boolean values or expressions, i.e., they are binary operators. The `and` operator evaluates an expression to `True` if both Boolean values are `True`; otherwise, it evaluates to `False`.

On the other hand, the or operator evaluates an expression to True if either of the two Boolean values is True. If both are False, it evaluates to False.

Expression	Evaluates to...
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

Expression	Evaluates to...
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

Unlike `and` and `or`, the `not` operator operates on only one Boolean value or expression. The `not` operator simply evaluates to the opposite Boolean value, i.e., its negation.

Expression	Evaluates to...
`not True`	`False`
`not False`	`True`

In [18]:

```
1 True and True
```

Out[18]:

```
True
```

In [20]:

```
1 True and False
```

Out[20]:

```
False
```

In [21]:

```
(5 > 4) and True
```

Out[21]:

```
True
```

In [22]:

```
False or (100 / 2 == 50)
```

Out[22]:

```
True
```

In [23]:

```
not not not not True
```

Out[23]:

```
True
```

In [24]:

```
2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2 and not 'Hej'.startswith('O'
#    True  and True           and   True          and True
```

Out[24]:

```
True
```

Checking Whether a value is in a List

Recall from the session on lists, that you have the `in` operator to check wether a value is a member of a list.

In [25]:

```
1 in [0, 1, 2, 3]
```

Out[25]:

```
True
```

In [38]:

```
1 not in [0, 1, 2, 3]
```

Out[38]:

```
False
```

`if` Statements

Simple if Statements

The simplest kind of if statement has one test and one action:

```
if conditional_test:
    statements
```

If the conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

In [46]:

```
if 5 < 4:
    print('Yep, right!')
    print('Still, right!')
else:
    print('Not true')
```

```
Not true
```

OBS! Remember intendation to match your intents.

In [47]:

```
if 5 < 4:
    print('Yep, right!')
print('Still, right!')
```

```
Still, right!
```

`if-else` Statements

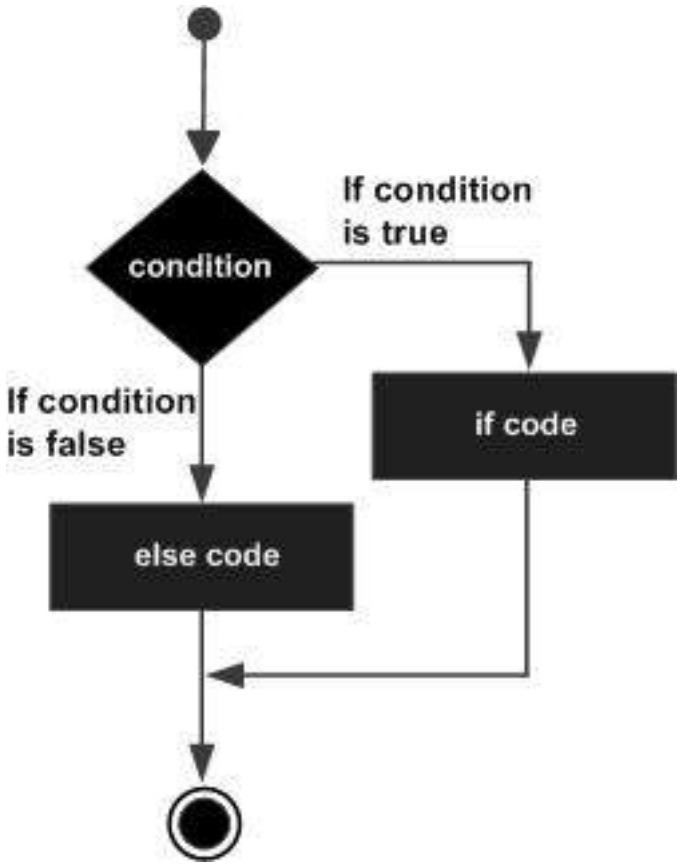
An if-else block is similar to a simple if statement, but the else statement allows you to define an action or set of actions that are executed when the conditional test fails.

In [48]:

```
title = 'Moby-Dick; or, the Whale'

if 'Moby-Dick' in title:
    print('by Herman Melville')
else:
    print('hmm, I do not know the author.')
```

```
by Herman Melville
```



The if-elif-else Chain

Often, you'll need to test more than two possible situations, and to evaluate these you can use Python's `if - elif - else` syntax. Python executes only one block in an if-elif-else chain. It runs each conditional test in order until one passes. When a test passes, the code following that test is executed and subsequent tests are skipped. You can use as many elif blocks in your code as you like.

Python does not require an else block at the end of an `if - elif` chain. Sometimes an else block is useful; sometimes it is clearer to use an additional `elif` statement that catches the specific condition of interest.

The `else` block is a catchall statement. It matches any condition that was not matched by a specific if or `elif` test, and that can sometimes include invalid or even malicious data. If you have a specific final condition you are testing for, consider using a final `elif` block and omit the else block. As a result, you will gain extra confidence that your code will run only under the correct conditions.

In [55]:

```
year = 1850

if year == 1851:
    message = 'First print.'
elif year == 1855:
    message = 'Second print.'
elif year == 1863:
    message = 'Third print.'
elif year == 1871:
    message = 'Fourth print.'
else:
    message = 'Hmm, I do not know this year...'

print(message)
```

```
Hmm, I do not know this year...
```

What is the difference between these two programs?

```
number = 10
if number > 0:
    print('Number is bigger than 0')
if number > 5:
    print('Number is bigger than 5')
```

```
number = 10
if number > 0:
    print('Number is bigger than 0')
elif number > 5:
    print('Number is bigger than 5')
```

What is the the following program doing?

```
name = input('Enter smart person here: ')
if name == 'John Locke':
    print('Capitalism is great!')
elif name == 'Karl Marx':
    print('Capitalism is evil!')
elif name == 'John M. Keynes':
    print('Capitalism is good when controlled')
else:
    print('Capitalism is a system for exchanging values')
```

Checking that a List is not Empty

Empty lists are `False` and non-empty lists will be `True`. This can be useful to quickly discover whether a list is empty or not:

```
library = []

if library:
    print('List is not empty.')  # If we reach this point, library is NOT empty
```

is equivalent to:

```
if library != []:
    print('List is not empty.')
```

However, the former is more pythonic.

In [ ]:

```
library = []

if library:
    print('List is not empty.')
else:
    print('List is empty.')
```

In [56]:

```
bool([])
```

Out[56]:

```
False
```

In [57]:

```
bool([1, 2, 3])
```

Out[57]:

```
True
```