01-quick-refresher-basic-review

June 14, 2023

0.1 Multi-Line Statements and Strings

Certain physical newlines are ignored in order to form a complete logical line of code.

Implicit Examples

[1]:
```
a = [1,
     2,
     3]
```

[2]:
```
a
```

[2]:
```
[1, 2, 3]
```

You may also add comments to the end of each physical line:

[3]:
```
a = [1, #first element
     2, #second element
     3, #third element
     ]
```

[4]:
```
a
```

[4]:
```
[1, 2, 3]
```

Note if you do use comments, you must close off the collection on a new line.

i.e. the following will not work since the closing] is actually part of the comment:

[5]:
```
a = [1, # first element
     2 #second element]
```

```
  Cell In[5], line 2
    2 #second element]
                      ^
SyntaxError: incomplete input
```

This works the same way for tuples, sets, and dictionaries.

[6]:
```
a = (1, # first element
    2, #second element
    3, #third element
    )
```

[7]:
```
a
```

[7]:
```
(1, 2, 3)
```

[8]:
```
a = {1, # first element
    2, #second element
    }
```

[9]:
```
a
```

[9]:
```
{1, 2}
```

[10]:
```
a = {'key1': 'value1', #comment,
    'key2': #comment
    'value2' #comment
    }
```

[11]:
```
a
```

[11]:
```
{'key1': 'value1', 'key2': 'value2'}
```

We can also break up function arguments and parameters:

[12]:
```
def my_func(a, #some comment
            b, c):
    print(a, b, c)
```

[13]:
```
my_func(10, #comment
        20, #comment
        30)
```

```
10 20 30
```

Explicit Examples You can use the \ character to explicitly create multi-line statements.

[14]:
```
a = 10
b = 20
c = 30
if a > 5 \
    and b > 10 \
    and c > 20:
    print('yes!!')
```

```
yes!!
```

The identation in continued-lines does not matter:

[15]:
```
a = 10
b = 20
c = 30
if a > 5 \
    and b > 10 \
        and c > 20:
    print('yes!!')
```

```
yes!!
```

Multi-Line Strings You can create multi-line strings by using triple delimiters (single or double quotes)

[16]:
```
a = '''this is
a multi-line string'''
```

[17]:
```
print(a)
```

```
this is
a multi-line string
```

Note how the newline character we typed in the multi-line string was preserved. Any character you type is preserved. You can also mix in escaped characters line any normal string.

[18]:
```
a = """some items:\n
    1. item 1
    2. item 2"""
```

[19]:
```
print(a)
```

```
some items:

    1. item 1
    2. item 2
```

Be careful if you indent your multi-line strings - the extra spaces are preserved!

[20]:
```
def my_func():
    a = '''a multi-line string
    that is actually indented in the second line'''
    return a
```

[21]:
```
print(my_func())
```

```
a multi-line string
    that is actually indented in the second line
```

[22]:
```
def my_func():
    a = '''a multi-line string
that is not indented in the second line'''
    return a
```

[23]:
```
print(my_func())
```

```
a multi-line string
that is not indented in the second line
```

Note that these multi-line strings are **not** comments - they are real strings and, unlike comments, are part of your compiled code. They are however sometimes used to create comments, such as `docstrings`, that we will cover later in this course.

In general, use `#` to comment your code, and use multi-line strings only when actually needed (like for docstrings).

Also, there are no multi-line comments in Python. You simply have to use a `#` on every line.

[24]:
```
# this is
#    a multi-line
#    comment
```

The following works, but the above formatting is preferrable.

[25]:
```
# this is
    # a multi-line
    # comment
```

0.1.1 Conditionals

A conditional is a construct that allows you to branch your code based on conditions being met (or not)

This is achieved using **if**, **elif** and **else** or the **ternary operator** (aka conditional expression)

[26]:
```
a = 2
if a < 3:
    print('a < 3')
else:
    print('a >= 3')
```

```
a < 3
```

if statements can be nested:

[27]:
```
a = 15

if a < 5:
    print('a < 5')
else:
```

```
    if a < 10:
        print('5 <= a < 10')
    else:
        print('a >= 10')
```

```
a >= 10
```

But the **elif** statement provides far better readability:

[28]:
```
a = 15
if a < 5:
    print('a < 5')
elif a < 10:
    print('5 <= a < 10')
else:
    print('a >= 10')
```

```
a >= 10
```

In Python, **elif** is the closest you'll find to the switch/case statement available in some other languages.

Python also provides a conditional expression (ternary operator):

X if (condition) else Y

returns (and evaluates) X if (condition) is True, otherwise returns (and evaluates) Y

[29]:
```
a = 5
res = 'a < 10' if a < 10 else 'a >= 10'
print(res)
```

```
a < 10
```

[30]:
```
a = 15
res = 'a < 10' if a < 10 else 'a >= 10'
print(res)
```

```
a >= 10
```

Note that **X** and **Y** can be any expression, not just literal values:

[31]:
```
def say_hello():
    print('Hello!')

def say_goodbye():
    print('Goodbye!')
```

[32]:
```
a = 5
say_hello() if a < 10 else say_goodbye()
```

```
Hello!
```

[33]:
```
a = 15
say_hello() if a < 10 else say_goodbye()
```

```
Goodbye!
```

0.1.2 Functions

Python has many built-in functions and methods we can use

Some are available by default:

[34]:
```
s = [1, 2, 3]
len(s)
```

[34]:
```
3
```

While some need to be imported:

[35]:
```
from math import sqrt
```

[36]:
```
sqrt(4)
```

[36]:
```
2.0
```

Entire modules can be imported:

[37]:
```
import math
```

[38]:
```
math.exp(1)
```

[38]:
```
2.718281828459045
```

We can define our own functions:

[39]:
```
def func_1():
    print('running func1')
```

[40]:
```
func_1()
```

```
running func1
```

Note that to "call" or "invoke" a function we need to use the ().

Simply using the function name without the () refers to the function, but does not call it:

[41]:
```
func_1
```

[41]:
```
<function __main__.func_1()>
```

We can also define functions that take parameters:

[42]:
```
def func_2(a, b):
    return a * b
```

Note that **a** and **b** can be any type (this is an example of polymorphism - which we will look into more detail later in this course).

But the function will fail to run if **a** and **b** are types that are not "compatible" with the ***** operator:

[43]:
```
func_2(3, 2)
```

[43]:
```
6
```

[44]:
```
func_2('a', 3)
```

[44]:
```
'aaa'
```

[45]:
```
func_2([1, 2, 3], 2)
```

[45]:
```
[1, 2, 3, 1, 2, 3]
```

[46]:
```
func_2('a', 'b')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[46], line 1
----> 1 func_2('a', 'b')

Cell In[42], line 2, in func_2(a, b)
      1 def func_2(a, b):
----> 2     return a * b

TypeError: can't multiply sequence by non-int of type 'str'
```

It is possible to use **type annotations**:

[47]:
```
def func_3(a: int, b:int):
    return a * b
```

[48]:
```
func_3(2, 3)
```

[48]:
```
6
```

[49]:
```
func_3('a', 2)
```

[49]:
```
'aa'
```

But as you can see, these do not enforce a data type! They are simply metadata that can be used by external libraries, and many IDE's.

Functions are objects, just like integers are objects, and they can be assigned to variables just as an integer can:

[50]:
```
my_func = func_3
```

[51]:
```
my_func('a', 2)
```

[51]:
```
'aa'
```

Functions **must** always return something. If you do not specify a return value, Python will automatically return the **None** object:

[52]:
```
def func_4():
    # does something but does not return a value
    a = 2
```

[53]:
```
res = func_4()
```

[54]:
```
print(res)
```

```
None
```

The **def** keyword is an executable piece of code that creates the function (an instance of the **function** class) and essentially assigns it to a variable name (the function **name**).

Note that the function is defined when **def** is reached, but the code inside it is not evaluated until the function is called.

This is why we can define functions that call other functions defined later - as long as we don't call them before all the necessary functions are defined.

For example, the following will work:

[55]:
```
def fn_1():
    fn_2()

def fn_2():
    print('Hello')

fn_1()
```

```
Hello
```

But this will not work:

[56]:
```
def fn_3():
    fn_4()

fn_3()
```

```
def fn_4():
    print('Hello')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[56], line 4
      1 def fn_3():
      2     fn_4()
----> 4 fn_3()
      6 def fn_4():
      7     print('Hello')

Cell In[56], line 2, in fn_3()
      1 def fn_3():
----> 2     fn_4()

NameError: name 'fn_4' is not defined
```

We also have the **lambda** keyword, that also creates a new function, but does not assign it to any specific name - instead it just returns the function object - which we can, if we wish, assign to a variable ourselves:

[57]:
```
func_5 = lambda x: x**2
```

[58]:
```
func_5
```

[58]:
```
<function __main__.<lambda>(x)>
```

[59]:
```
func_5(2)
```

[59]:
```
4
```

We'll examine lambdas in more detail later in this course.

0.1.3 While Loops

The **while** loop is a way to repeatg a block of code as long as a specified condition is met.

`while <exp is true>:     code block`

[60]:
```
i = 0
while i < 5:
    print(i)
    i += 1
```

```
0
1
```

```
2
3
4
```

Note that there is no guarantee that a **while** loop will execute at all, not even once, because the condition is tested **before** the loop runs.

[61]:
```
i = 5
while i < 5:
    print(i)
    i += 1
```

Some languages have a concept of a while loop that is guaranteed to execute at least once:

```
do     code block while <exp is true>
```

There is no such thing in Python, but it's easy enough to write code that works that way.

We create an infinite loop and test the condition inside the loop and break out of the loop when the condition becomes false:

[62]:
```
i = 5

while True:
    print(i)
    if i >= 5:
        break
```

```
5
```

As you can see the loop executed once (and will always execute at least once, no matter the starting value of i.)

This is a standard pattern and can be useful in a variety of scenarios.

A simple example might be getting repetitive user input until the user performs and action or provides some specific value. For example, suppose we want to use the console to let users enter their name. We just want to make sure their name is at least 2 characters long, contains printable characters only, and only contains alphabetic characters: We might try it this way:

[64]:
```
min_length = 2

name = input('Please enter your name:')

while not(len(name) >= min_length  and name.isprintable() and name.isalpha()):
    name = input('Please enter your name:')

print('Hello, {0}'.format(name))
```

```
Please enter your name: haresh

Hello, haresh
```

This works just fine, but notice that we had to write the code to elicit user input **twice** in our code. This is not good practice, and we can easily clean this up as follows:

[66]:
```
min_length = 2

while True:
    name = input('Please enter your name:')
    if len(name) >= min_length  and name.isprintable() and name.isalpha():
        break

print('Hello, {0}'.format(name))
```

```
Please enter your name: haresh
```

```
Hello, haresh
```

We saw how the **break** statement exits the **while** loop and execution resumes on the line immediately after the while code block.

Sometimes, we just want to cut the current iteration short, but continue looping, without exiting the loop itself.

This is done using the **continue** statement:

[67]:
```
a = 0
while a < 10:
    a += 1
    if a % 2:
        continue
    print(a)
```

```
2
4
6
8
10
```

Note that there are much better ways of doing this! We'll cover that in later videos (comprehensions, generators, etc)

The **while** loop also can be used with an **else** clause!!

The **else** is executed if the while loop terminated without hitting a **break** statement (we say the loop terminated **normally**)

Suppose we want to test if some value is present in some list, and if not we want to append it to the list (again there are better ways of doing this):

First, here's how we might do it without the benefit of the **else** clause:

[68]:
```
l = [1, 2, 3]
val = 10
```

```
found = False
idx = 0
while idx < len(l):
    if l[idx] == val:
        found = True
        break
    idx += 1

if not found:
    l.append(val)
print(l)
```

```
[1, 2, 3, 10]
```

Using the **else** clause is easier:

[69]:
```
l = [1, 2, 3]
val = 10

idx = 0
while idx < len(l):
    if l[idx] == val:
        break
    idx += 1
else:
    l.append(val)

print(l)
```

```
[1, 2, 3, 10]
```

[70]:
```
l = [1, 2, 3]
val = 3

idx = 0
while idx < len(l):
    if l[idx] == val:
        break
    idx += 1
else:
    l.append(val)

print(l)
```

```
[1, 2, 3]
```

0.1.4 Loop Break and Continue inside a Try...Except...Finally

Recall that in a `try` statement, the `finally` clause always runs:

[71]:
```
a = 10
b = 1
try:
    a / b
except ZeroDivisionError:
    print('division by 0')
finally:
    print('this always executes')
```

```
this always executes
```

[72]:
```
a = 10
b = 0
try:
    a / b
except ZeroDivisionError:
    print('division by 0')
finally:
    print('this always executes')
```

```
division by 0
this always executes
```

So, what happens when using a `try` statement within a `while` loop, and a `continue` or `break` statement is encountered?

[73]:
```
a = 0
b = 2

while a < 3:
    print('-------------')
    a += 1
    b -= 1
    try:
        res = a / b
    except ZeroDivisionError:
        print('{0}, {1} - division by 0'.format(a, b))
        res = 0
        continue
    finally:
        print('{0}, {1} - always executes'.format(a, b))

    print('{0}, {1} - main loop'.format(a, b))
```

```
-------------
1, 1 - always executes
1, 1 - main loop
-------------
2, 0 - division by 0
```

```
2, 0 - always executes
-------------
3, -1 - always executes
3, -1 - main loop
```

As you can see in the above result, the `finally` code still executed, even though the current iteration was cut short with the `continue` statement.

This works the same with a `break` statement:

[74]:
```
a = 0
b = 2

while a < 3:
    print('-------------')
    a += 1
    b -= 1
    try:
        res = a / b
    except ZeroDivisionError:
        print('{0}, {1} - division by 0'.format(a, b))
        res = 0
        break
    finally:
        print('{0}, {1} - always executes'.format(a, b))

    print('{0}, {1} - main loop'.format(a, b))
```

```
-------------
1, 1 - always executes
1, 1 - main loop
-------------
2, 0 - division by 0
2, 0 - always executes
```

We can even combine all this with the `else` clause:

[75]:
```
a = 0
b = 2

while a < 3:
    print('-------------')
    a += 1
    b -= 1
    try:
        res = a / b
    except ZeroDivisionError:
        print('{0}, {1} - division by 0'.format(a, b))
        res = 0
        break
```

```
    finally:
        print('{0}, {1} - always executes'.format(a, b))

    print('{0}, {1} - main loop'.format(a, b))
else:
    print('\n\nno errors were encountered!')
```

```
-------------
1, 1 - always executes
1, 1 - main loop
-------------
2, 0 - division by 0
2, 0 - always executes
```

[76]:
```
a = 0
b = 5

while a < 3:
    print('-------------')
    a += 1
    b -= 1
    try:
        res = a / b
    except ZeroDivisionError:
        print('{0}, {1} - division by 0'.format(a, b))
        res = 0
        break
    finally:
        print('{0}, {1} - always executes'.format(a, b))

    print('{0}, {1} - main loop'.format(a, b))
else:
    print('\n\nno errors were encountered!')
```

```
-------------
1, 4 - always executes
1, 4 - main loop
-------------
2, 3 - always executes
2, 3 - main loop
-------------
3, 2 - always executes
3, 2 - main loop


no errors were encountered!
```

[ ]:

0.1.5 The For Loop

An **iterable** is something can be iterated over. :-)

Maybe a better non-circular way to define iterable is to think of it as a collection of things that can be accessed one at a time.

In Python, an **iterable** has a very specific meaning: an iterable is an **object** capable of returning its members one at a time.

Many objects in Python are iterable: lists, strings, file objects and many more.

The **for** keyword can be used to iterate an iterable.

If you come with a background in another programming language, you have probably seen **for** loops defined this way:

```
for (int i=0; i < 5; i++) {     //code block }
```

This form of the **for** loop is simply a *repetition*, very similar to a **while** loop - in fact it is equivalent to what we could write in Python as follows:

[77]:
```
i = 0
while i < 5:
    #code block
    print(i)
    i += 1
i = None
```

```
0
1
2
3
4
```

But that's **NOT** what the **for** statement does in Python - the **for** statement is a way to **iterate** over iterables, and has nothing to do with the **for** loop we just saw. The closest equivalent we have in Python is the **where** loop written as above.

To use the **for** loop in Python, we **require** an iterable object to work with.

A simple iterable object is generated via the `range()` function

[78]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Although this might seem like the closest approximation in Python to the standard C-style for loop we saw earlier, it's not really - we are iterating over an iterable object which is quite different.

Many objects are iterable in Python:

[79]:
```
for x in [1, 2, 3]:
    print(x)
```

```
1
2
3
```

[80]:
```
for x in 'hello':
    print(x)
```

```
h
e
l
l
o
```

[81]:
```
for x in ('a', 'b', 'c'):
    print(x)
```

```
a
b
c
```

When we iterate over an iterable, each iteration returns the "next" value (or object) in the iterable:

[82]:
```
for x in [(1, 2), (3, 4), (5, 6)]:
    print(x)
```

```
(1, 2)
(3, 4)
(5, 6)
```

We can even assign the individual tuple values to specific named variables:

[83]:
```
for i, j in [(1, 2), (3, 4), (5, 6)]:
    print(i, j)
```

```
1 2
3 4
5 6
```

We will cover iterables in a lot more detail later in this course.

The **break** and **continue** statements work just as well in **for** loops as they do in **where** loops:

[84]:
```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

```
0
1
2
4
```

[85]:
```
for i in range(5):
    if i == 3:
        break
    print(i)
```

```
0
1
2
```

The **for** loop, like the **while** loop, also supports an **else** clause which is executed if and only if the loop terminates normally (i.e. did not exit because of a **break** statement)

[86]:
```
for i in range(1, 5):
    print(i)
    if i % 7 == 0:
        print('multiple of 7 found')
        break
else:
    print('No multiples of 7 encountered')
```

```
1
2
3
4
No multiples of 7 encountered
```

[87]:
```
for i in range(1, 8):
    print(i)
    if i % 7 == 0:
        print('multiple of 7 found')
        break
else:
    print('No multiples of 7 encountered')
```

```
1
2
3
4
5
```

```
6
7
multiple of 7 found
```

Similarly to the **where** loop, **break** and **continue** work just the same in the context of a **try** statement's **finally** clause.

[88]:
```
for i in range(5):
    print('--------------------')
    try:
        10 / (i - 3)
    except ZeroDivisionError:
        print('divided by 0')
        continue
    finally:
        print('always runs')
    print(i)
```

```
--------------------
always runs
0
--------------------
always runs
1
--------------------
always runs
2
--------------------
divided by 0
always runs
--------------------
always runs
4
```

There are a number of standard techniques to iterate over iterables:

[89]:
```
s = 'hello'
for c in s:
    print(c)
```

```
h
e
l
l
o
```

But sometimes, for indexable iterable types (e.g. sequences), we want to also know the index of the item in the loop:

[90]:
```
s = 'hello'
i = 0
for c in s:
    print(i, c)
    i += 1
```

```
0 h
1 e
2 l
3 l
4 o
```

Slightly better approach might be:

[91]:
```
s = 'hello'

for i in range(len(s)):
    print(i, s[i])
```

```
0 h
1 e
2 l
3 l
4 o
```

or even better:

[92]:
```
s = 'hello'

for i, c in enumerate(s):
    print(i, c)
```

```
0 h
1 e
2 l
3 l
4 o
```

We'll come back to all these iteration techniques in a lot more detail throughout this course.

0.1.6 Custom Classes

We'll cover classes in a lot of detail in this course, but for now you should have at least some understanding of classes in Python and how to create them.

To create a custom class we use the `class` keyword, and we can initialize class attributes in the special method `__init__`.

[93]:
```
class Rectangle:
    def __init__(self, width, height):
```

```
        self.width = width
        self.height = height
```

We create **instances** of the `Rectangle` class by calling it with arguments that are passed to the `__init__` method as the second and third arguments. The first argument (`self`) is automatically filled in by Python and contains the object being created.

Note that using `self` is just a convention (although a good one, and you shgoudl use it to make your code more understandable by others), you could really call it whatever (valid) name you choose.

But just because you can does not mean you should!

[94]:
```
r1 = Rectangle(10, 20)
r2 = Rectangle(3, 5)
```

[95]:
```
r1.width
```

[95]: 10

[96]:
```
r2.height
```

[96]: 5

`width` and `height` are attributes of the `Rectangle` class. But since they are just values (not callables), we call them **properties**.

Attributes that are callables are called **methods**.

You'll note that we were able to retrieve the `width` and `height` attributes (properties) using a dot notation, where we specify the object we are interested in, then a dot, then the attribute we are interested in.

We can add callable attributes to our class (methods), that will also be referenced using the dot notation.

Again, we will create instance methods, which means the method will require the first argument to be the object being used when the method is called.

[97]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(the_referenced_object):
        return 2 * (the_referenced_object.width + the_referenced_object.height)
```

[98]:
```
r1 = Rectangle(10, 20)
```

[99]:
```
r1.area()
```

[99]:
```
200
```

When we ran the above line of code, our object was `r1`, so when `area` was called, Python in fact called the method `area` in the Rectangle class automatically passing `r1` to the `self` parameter.

This is why we can use a name other than self, such as in the perimeter method:

[100]:
```
r1.perimeter()
```

[100]:
```
60
```

Again, I'm just illustrating a point, don't actually do that!

[101]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)
```

[102]:
```
r1 = Rectangle(10, 20)
```

Python defines a bunch of **special** methods that we can use to give our classes functionality that resembles functionality of built-in and standard library objects.

Many people refer to them as *magic* methods, but there's nothing magical about them - unlike magic, they are well documented and understood!!

These **special** methods provide us an easy way to overload operators in Python.

For example, we can obtain the string representation of an integer using the built-in `str` function:

[103]:
```
str(10)
```

[103]:
```
'10'
```

What happens if we try this with our Rectangle object?

[104]:
```
str(r1)
```

[104]:
```
'<__main__.Rectangle object at 0x000002113C456DD0>'
```

Not exactly what we might have expected. On the other hand, how is Python supposed to know how to display our rectangle as a string?

We could write a method in the class such as:

[105]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

    def to_str(self):
        return 'Rectangle (width={0}, height={1})'.format(self.width, self.
 ↪height)
```

So now we could get a string from our object as follows:

[106]:
```
r1 = Rectangle(10, 20)
r1.to_str()
```

[106]:
```
'Rectangle (width=10, height=20)'
```

But of course, using the built-in `str` function still does not work:

[107]:
```
str(r1)
```

[107]:
```
'<__main__.Rectangle object at 0x000002113BEE8510>'
```

Does this mean we are out of luck, and anyone who writes a class in Python will need to provide some method to do this, and probably come up with their own name for the method too, maybe `to_str`, `make_string`, `stringify`, and who knows what else.

Fortunately, this is where these special methods come in. When we call `str(r1)`, Python will first look to see if our class (`Rectangle`) has a special method called `__str__`.

If the `__str__` method is present, then Python will call it and return that value.

There's actually another one called `__repr__` which is related, but we'll just focus on `__str__` for now.

[108]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

```
    def perimeter(self):
        return 2 * (self.width + self.height)

    def __str__(self):
        return 'Rectangle (width={0}, height={1})'.format(self.width, self.
↪height)
```

[109]:
```
r1 = Rectangle(10, 20)
```

[110]:
```
str(r1)
```

[110]:
```
'Rectangle (width=10, height=20)'
```

However, in Jupyter (and interactive console if you are using that), look what happens here:

[111]:
```
r1
```

[111]:
```
<__main__.Rectangle at 0x2113bf014d0>
```

As you can see we still get that default. That's because here Python is not converting `r1` to a string, but instead looking for a string *representation* of the object. It is looking for the `__repr__` method (which we'll come back to later).

[112]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

    def __str__(self):
        return 'Rectangle (width={0}, height={1})'.format(self.width, self.
↪height)

    def __repr__(self):
        return 'Rectangle({0}, {1})'.format(self.width, self.height)
```

[113]:
```
r1 = Rectangle(10, 20)
```

[114]:
```
print(r1)  # uses __str__
```

```
Rectangle (width=10, height=20)
```

[115]:
```
r1  # uses __repr__
```

[115]:
```
Rectangle(10, 20)
```

How about the comparison operators, such as == or <?

[116]:
```
r1 = Rectangle(10, 20)
r2 = Rectangle(10, 20)
```

[117]:
```
r1 == r2
```

[117]:
```
False
```

As you can see, Python does not consider `r1` and `r2` as equal (using the == operator). Again, how is Python supposed to know that two Rectangle objects with the same height and width should be considered equal?

We just need to tell Python how to do it, using the special method `__eq__`.

[118]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

    def __str__(self):
        return 'Rectangle (width={0}, height={1})'.format(self.width, self.
 ↪height)

    def __repr__(self):
        return 'Rectangle({0}, {1})'.format(self.width, self.height)

    def __eq__(self, other):
        print('self={0}, other={1}'.format(self, other))
        if isinstance(other, Rectangle):
            return (self.width, self.height) == (other.width, other.height)
        else:
            return False
```

[119]:
```
r1 = Rectangle(10, 20)
r2 = Rectangle(10, 20)
```

[120]:
```
r1 is r2
```

[120]:
```
False
```

[121]:
```
r1 == r2
```

```
self=Rectangle (width=10, height=20), other=Rectangle (width=10, height=20)
```

[121]:
```
True
```

[122]:
```
r3 = Rectangle(2, 3)
```

[123]:
```
r1 == r3
```

```
self=Rectangle (width=10, height=20), other=Rectangle (width=2, height=3)
```

[123]:
```
False
```

And if we try to compare our Rectangle to a different type:

[124]:
```
r1 == 100
```

```
self=Rectangle (width=10, height=20), other=100
```

[124]:
```
False
```

Let's remove that print statement - I only put that in so you could see what the arguments were, in practice you should avoid side effects.

[125]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

    def __str__(self):
        return 'Rectangle (width={0}, height={1})'.format(self.width, self.
 ↪height)

    def __repr__(self):
        return 'Rectangle({0}, {1})'.format(self.width, self.height)

    def __eq__(self, other):
        if isinstance(other, Rectangle):
            return (self.width, self.height) == (other.width, other.height)
        else:
            return False
```

What about <, >, <=, etc.?

Again, Python has special methods we can use to provide that functionality.

These are methods such as __lt__, __gt__, __le__, etc.

[126]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

    def __str__(self):
        return 'Rectangle (width={0}, height={1})'.format(self.width, self.
 ↪height)

    def __repr__(self):
        return 'Rectangle({0}, {1})'.format(self.width, self.height)

    def __eq__(self, other):
        if isinstance(other, Rectangle):
            return (self.width, self.height) == (other.width, other.height)
        else:
            return False

    def __lt__(self, other):
        if isinstance(other, Rectangle):
            return self.area() < other.area()
        else:
            return NotImplemented
```

[127]:
```
r1 = Rectangle(100, 200)
r2 = Rectangle(10, 20)
```

[128]:
```
r1 < r2
```

[128]:
```
False
```

[129]:
```
r2 < r1
```

[129]:
```
True
```

What about >?

[130]:
```
r1 > r2
```

[130]:
```
True
```

How did that work? We did not define a `__gt__` method.

Well, Python cleverly decided that since `r1 > r2` was not implemented, it would give

`r2 < r1`

a try. And since, `__lt__` **is** defined, it worked!

Of course, `<=` is not going to magically work!

[131]:
```
r1 <= r2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[131], line 1
----> 1 r1 <= r2

TypeError: '<=' not supported between instances of 'Rectangle' and 'Rectangle'
```

If you come from a Java background, you are probably thinking that using "bare" properties (direct access), such as `height` and `width` is a terrible design idea.

It is for Java, but not for Python.

Although you can use bare properties in Java, if you ever need to intercept the getting or setting of a property, you will need to write a method (such as `getWidth` and `setWidth`. The problem is that if you used a bare `width` property for example, a lot of your code might be using `obj.width` (as we have been doing here). The instant you make the `width` private and instead implement getters and setters, you break your code. Hence one of the reasons why in Java we just write getters and setters for properties from the beginning.

With Python this is not the case - we can change any bare property into getters and setters without breaking the code that uses that bare property.

I'll show you a quick example here, but we'll come back to this topic in much more detail later.

Let's take our Rectangle class once again. I'll use a simplified version to keep the code short.

[132]:
```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def __repr__(self):
        return 'Rectangle({0}, {1})'.format(self.width, self.height)
```

[133]:
```
r1 = Rectangle(10, 20)
```

[134]:
```
r1.width
```

[134]:
```
10
```

[135]:
```
r1.width = 100
```

[136]:
```
r1
```

[136]:
```
Rectangle(100, 20)
```

As you saw we can *get* and *set* the `width` property directly.

But let's say after this code has been released for a while and users of our class have been using it (and specifically setting and getting the `width` and `height` attribute a lot), but now we want to make sure users cannot set a non-positive value (i.e. <= 0) for width (or height, but we'll focus on width as an example).

In a language like Java, we would implement `getWidth` and `setWidth` and make `width` private - which would break any code directly accessing the `width` property.

In Pytyhon we can use some special **decorators** (more on those later) to encapsulate our property getters and setters:

[137]:
```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    def __repr__(self):
        return 'Rectangle({0}, {1})'.format(self.width, self.height)

    @property
    def width(self):
        return self._width

    @width.setter
    def width(self, width):
        if width <= 0:
            raise ValueError('Width must be positive.')
        self._width = width

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, height):
        if height <= 0:
            raise ValueError('Height must be positive.')
```

```
        self._height = height
```

[138]:
```
r1 = Rectangle(10, 20)
```

[139]:
```
r1.width
```

[139]:
```
10
```

[140]:
```
r1.width = 100
```

[141]:
```
r1
```

[141]:
```
Rectangle(100, 20)
```

[142]:
```
r1.width = -10
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[142], line 1
----> 1 r1.width = -10

Cell In[137], line 16, in Rectangle.width(self, width)
     13 @width.setter
     14 def width(self, width):
     15     if width <= 0:
---> 16         raise ValueError('Width must be positive.')
     17     self._width = width

ValueError: Width must be positive.
```

There are more things we should do to properly implement all this, in particular we should also be checking the positive and negative values during the `__init__` phase. We do so by using the accessor methods for height and width:

[143]:
```
class Rectangle:
    def __init__(self, width, height):
        self._width = None
        self._height = None
        # now we call our accessor methods to set the width and height
        self.width = width
        self.height = height

    def __repr__(self):
        return 'Rectangle({0}, {1})'.format(self.width, self.height)

    @property
```

```
    def width(self):
        return self._width

    @width.setter
    def width(self, width):
        if width <= 0:
            raise ValueError('Width must be positive.')
        self._width = width

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, height):
        if height <= 0:
            raise ValueError('Height must be positive.')
        self._height = height
```

[144]:
```
r1 = Rectangle(0, 10)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[144], line 1
----> 1 r1 = Rectangle(0, 10)

Cell In[143], line 6, in Rectangle.__init__(self, width, height)
      4 self._height = None
      5 # now we call our accessor methods to set the width and height
----> 6 self.width = width
      7 self.height = height

Cell In[143], line 19, in Rectangle.width(self, width)
     16 @width.setter
     17 def width(self, width):
     18     if width <= 0:
---> 19         raise ValueError('Width must be positive.')
     20     self._width = width

ValueError: Width must be positive.
```

There more we should be doing, like checking that the width and height being passed in are numeric types, and so on. Especially during the `__init__` phase - we would rather raise an exception when the object is being created rather than delay things and raise an exception when the user calls some method like `area` - that way the exception will be on the line that creates the object - makes debugging much easier!

There are many more of these special methods, and we'll look in detail at them later in this course.