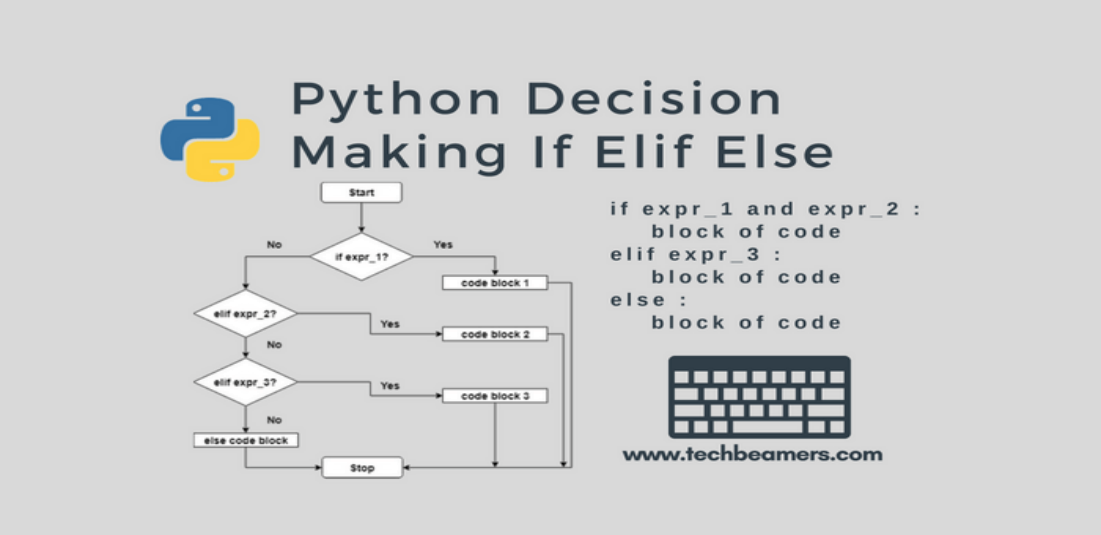
Day_4

Booleans and Conditionals



Booleans:

Python has a type of variable called bool. It has two possible values: True and False.

4.1.Basic Boolean.py

```
x = True
print(x)
print(type(x))
```

Output:
True
<class 'bool'>

Rather than putting True or False directly in our code, we usually get boolean values from **boolean operators**. These are operators that answer yes/no questions. We'll go through some of these operators below.

Comparison Operations

Operation	Description		Operation	Description
a == b	a equal to b		a != b	a not equal to b
a < b	a less than b		a > b	a greater than b
a <= b	a less than or equal to b		a >= b	a greater than or equal to b

4.2. Boolean Comparison.py

```
def can_run_for_president(age):
    """Can someone of the given age run for president in the US?"""
    # The US Constitution says you must be at least 35 years old
    return age >= 35

print("Can a 19-year-old run for president?", can_run_for_president(19))
print("Can a 45-year-old run for president?", can_run_for_president(45))
```

Output:
Can a 19-year-old run for president? False
Can a 45-year-old run for president? True

Comparisons frequently work like you'd hope

```
3.0 == 3
```
Output: True

But sometimes they can be tricky

```
'3' == 3
```
Output: False

Comparison operators can be combined with the arithmetic operators we've already seen to express a virtually limitless range of mathematical tests. For example, we can check if a number is odd by checking that the modulus with 2 returns 1:

4.3. odd or not.py

```
def is_odd(n):
    return (n % 2) == 1

print("Is 100 odd?", is_odd(100))
print("Is -1 odd?", is_odd(-1))
```

Output:
Is 100 odd? False
Is -1 odd? True

Remember to use == instead of = when making comparisons. If you write n == 2 you are asking about the value of n. When you write n = 2 you are changing the value of n.

Combining Boolean Values

You can combine boolean values using the standard concepts of "and", "or", and "not". In fact, the words to do this are: and, or, and not.
With these, we can make our can_run_for_president function more accurate.

4.4. Be president or not.py

```
def can_run_for_president(age, is_natural_born_citizen):
    """Can someone of the given age and citizenship status run for president in the US?"""
    # The US Constitution says you must be a natural born citizen *and* at least 35 years old
    return is_natural_born_citizen and (age >= 35)

print(can_run_for_president(19, True))
print(can_run_for_president(55, False))
print(can_run_for_president(55, True))
```

Output:
False
False
True

Quick, can you guess the value of this expression?

True **or** True **and** False
To answer this, you'd need to figure out the order of operations.
For example, and is evaluated before or. That's why the first expression above is True. If we evaluated it from left to right, we would have calculated True or True first (which is True), and then taken the and of that result with False, giving a final value of False.
For example, consider the following expression:

```
prepared_for_weather = have_umbrella or rain_level < 5 and have_hood or not rain_level > 0 and is_workday
```

I'm trying to say that I'm safe from today's weather....

- if I have an umbrella...
- or if the rain isn't too heavy and I have a hood...
- otherwise, I'm still fine unless it's raining *and* it's a workday

But not only is my Python code hard to read, it has a bug. We can address both problems by adding some parentheses:

```
prepared_for_weather = have_umbrella or (rain_level < 5 and have_hood) or not (rain_level > 0 and is_workday)
```

You can add even more parentheses if you think it helps readability:

```
prepared_for_weather = have_umbrella or ((rain_level < 5) and have_hood) or (not (rain_level > 0 and is_workday))
```

We can also split it over multiple lines to emphasize the 3-part structure described above:

```
prepared_for_weather = (
    have_umbrella
    or ((rain_level < 5) and have_hood)
    or (not (rain_level > 0 and is_workday))
```

)

Conditionals

Booleans are most useful when combined with *conditional statements*, using the keywords if, elif, and else.
Conditional statements, often referred to as *if-then* statements, let you control what pieces of code are run based on the value of some Boolean condition. Here's an example:

4.5. positive or negative.py

```
def inspect(x):
    if x == 0:
        print(x, "is zero")
    elif x > 0:
        print(x, "is positive")
    elif x < 0:
        print(x, "is negative")
    else:
        print(x, "is unlike anything I've ever seen...")

inspect(0)
inspect(-15)
```

Output:
0 is zero
-15 is negative

The if and else keywords are often used in other languages; its more unique keyword is elif, a contraction of "else if". In these conditional clauses, elif and else blocks are optional; additionally, you can include as many elif statements as you would like. Note especially the use of colons (:) and whitespace to denote separate blocks of code. This is similar to what happens when we define a function - the function header ends with :, and the following line is indented with 4 spaces. All subsequent indented lines belong to the body of the function, until we encounter an unindented line, ending the function definition.

4.6. if-else print method.py

```
def f(x):
    if x > 0:
        print("Only printed when x is positive; x =", x)
        print("Also only printed when x is positive; x =", x)
    print("Always printed, regardless of x's value; x =", x)

f(1)
f(0)
```

Output:
Only printed when x is positive; x = 1
Also only printed when x is positive; x = 1
Always printed, regardless of x's value; x = 1

Always printed, regardless of x's value; x = 0

Boolean conversion

We've seen int(), which turns things into ints, and float(), which turns things into floats, so you might not be surprised to hear that Python has a bool() function which turns things into bools.

4.7.Boolean print.py

```
print(bool(1)) # all numbers are treated as true, except 0
print(bool(0))
print(bool("asf")) # all strings are treated as true, except the empty string ""
print(bool(""))
# Generally empty sequences (strings, lists, and other types we've yet to see like lists
and tuples)
# are "falsey" and the rest are "truthy"
```

Output:
True
False
True
False

We can use non-boolean objects in if conditions and other places where a boolean would be expected. Python will implicitly treat them as their corresponding boolean value:

```
if 0:
   print(0)
elif "spam":
   print("spam")
```

Output: spam

The candy-sharing friends Alice, Bob and Carol tried to split candies evenly. For the sake of their friendship, any candies left over would be smashed. For example, if they collectively bring home 91 candies, they'll take 30 each and smash 1.
Below is a simple function that will calculate the number of candies to smash for *any* number of total candies.

4.8.Candy_smash.py

```
def to_smash(total_candies):
   if total_candies == 1:
      print("Splitting", total_candies, "candy")
   else:
       print("Splitting", total_candies, "candies")
   return total_candies % 3
```

```
to_smash(91)
to_smash(1)
```

Here's a slightly more succinct solution using a conditional expression:

```
print("Splitting", total_candies, "candy" if total_candies == 1 else "candies")
```

deciding whether we're prepared for the weather. I said that I'm safe from today's weather if...

- I have an umbrella...
- or if the rain isn't too heavy and I have a hood...
- otherwise, I'm still fine unless it's raining *and* it's a workday

The function below uses our first attempt at turning this logic into a Python expression. I claimed that there was a bug in that code. Can you find it?

4.9. Rainy_Day.py

```
def prepared_for_weather(have_umbrella, rain_level, have_hood, is_workday):
    return have_umbrella or rain_level < 5 and have_hood or not rain_level > 0 and
is_workday

have_umbrella = True
rain_level = 0.0
have_hood = True
is_workday = True

actual = prepared_for_weather(have_umbrella, rain_level, have_hood, is_workday)
print(actual)
```

Output: True

Your challenge is –the function returned True (but should have returned False).

Your challenge solution & explanation:

One example of a failing test case is:
have_umbrella **= False**
rain_level **=** 0.0
have_hood **= False**
is_workday **= False**
Clearly we're prepared for the weather in this case. It's not raining. Not only that, it's not a workday, so we don't even need to leave the house! But our function will return False on these inputs.
The key problem is that Python implictly parenthesizes the last part as:
(**not** (rain_level **>** 0)) **and** is_workday
Whereas what we were trying to express would look more like:
not (rain_level **>** 0 **and** is_workday)

The "and" and "or" boolean operators allow building complex boolean expressions, for example:

```
name = "John"
age = 23
if name == "John" and age == 23:
   print("Your name is John, and you are also 23 years old.")

if name == "John" or name == "Rick":
   print("Your name is either John or Rick.")
```

Output:
Your name is John, and you are also 23 years old.
Your name is either John or Rick.

The "in" operator could be used to check if a specified object exists within an iterable object container, such as a list:

```
name = "John"
if name in ["John", "Rick"]:
   print("Your name is either John or Rick.")
```

Output: Your name is either John or Rick.

Now, exercise section, its your turn: ☺

```
number = 16
second_number = 0
first_array = [1,2,3]
second_array = [1,2]

if number > 15:
   print("1")

if first_array:
   print("2")

if len(second_array) == 2:
   print("3")

if len(first_array) + len(second_array) == 5:
   print("4")

if first_array and first_array[0] == 1:
   print("5")

if not second_number:
   print("6")
```

output:
1
2
3
4
5
6

---THANK YOU--