Python: Code Challenges (Optional)

codecademy

or Operator

The Python `or` operator combines two Boolean expressions and evaluates to `True` if at least one of the expressions returns `True`. Otherwise, if both expressions are `False`, then the entire expression evaluates to `False`.

```
True or True      # Evaluates to True
True or False     # Evaluates to True
False or False    # Evaluates to False
1 < 2 or 3 < 1    # Evaluates to True
3 < 1 or 1 > 6    # Evaluates to False
1 == 1 or 1 < 2   # Evaluates to True
```

Comparison Operators

In Python, *relational operators* compare two values or expressions. The most common ones are:
- `<` less than
- `>` greater than
- `<=` less than or equal to
- `>=` greater than or equal too

If the relation is sound, then the entire expression will evaluate to `True`. If not, the expression evaluates to `False`.

```
a = 2
b = 3
a < b  # evaluates to True
a > b  # evaluates to False
a >= b # evaluates to False
a <= b # evaluates to True
a <= a # evaluates to True
```

`if` Statement

The Python `if` statement is used to determine the execution of code based on the evaluation of a Boolean expression.
- If the `if` statement expression evaluates to `True`, then the indented code following the statement is executed.
- If the expression evaluates to `False` then the indented code following the `if` statement is skipped and the program executes the next line of code which is indented at the same level as the `if` statement.

```
# if Statement

test_value = 100

if test_value > 1:
  # Expression evaluates to True
  print("This code is executed!")

if test_value > 1000:
  # Expression evaluates to False
  print("This code is NOT executed!")

print("Program continues at this point.")
```

else Statement

The Python else statement provides alternate code to execute if the expression in an if statement evaluates to False.
The indented code for the if statement is executed if the expression evaluates to True. The indented code immediately following the else is executed only if the expression evaluates to False. To mark the end of the else block, the code must be unindented to the same level as the starting if line.

```python
# else Statement


test_value = 50


if test_value < 1:
  print("Value is < 1")
else:
  print("Value is >= 1")


test_string = "VALID"


if test_string == "NOT_VALID":
  print("String equals NOT_VALID")
else:
  print("String equals something else!")
```

and Operator

The Python and operator performs a Boolean comparison between two Boolean values, variables, or expressions. If both sides of the operator evaluate to True then the and operator returns True. If either side (or both sides) evaluates to False, then the and operator returns False. A non-Boolean value (or variable that stores a value) will always evaluate to True when used with the and operator.

```python
True and True      # Evaluates to True
True and False     # Evaluates to False
False and False    # Evaluates to False
1 == 1 and 1 < 2   # Evaluates to True
1 < 2 and 3 < 1    # Evaluates to False
"Yes" and 100      # Evaluates to True
```

elif Statement

The Python elif statement allows for continued checks to be performed after an initial if statement. An elif statement differs from the else statement because another expression is provided to be checked, just as with the initial if statement.
If the expression is True, the indented code following the elif is executed. If the expression evaluates to False, the code can continue to an optional else statement. Multiple elif statements can be used following an initial if to perform a series of checks. Once an elif expression evaluates to True, no further elif statements are executed.

```python
# elif Statement


pet_type = "fish"


if pet_type == "dog":
  print("You have a dog.")
elif pet_type == "cat":
  print("You have a cat.")
elif pet_type == "fish":
  # this is performed
  print("You have a fish")
```

```
else:
    print("Not sure!")
```

Equal Operator ==

The equal operator, `==` , is used to compare two values, variables or expressions to determine if they are the same.
If the values being compared are the same, the operator returns True , otherwise it returns False .
The operator takes the data type into account when making the comparison, so a string value of "2" is *not* considered the same as a numeric value of 2 .

```python
# Equal operator

if 'Yes' == 'Yes':
    # evaluates to True
    print('They are equal')

if (2 > 1) == (5 < 10):
    # evaluates to True
    print('Both expressions give the same
result')

c = '2'
d = 2

if c == d:
    print('They are equal')
else:
    print('They are not equal')
```

Not Equals Operator !=

The Python not equals operator, `!=` , is used to compare two values, variables or expressions to determine if they are NOT the same. If they are NOT the same, the operator returns True . If they are the same, then it returns False .
The operator takes the data type into account when making the comparison so a value of 10 would NOT be equal to the string value "10" and the operator would return True . If expressions are used, then they are evaluated to a value of True or False before the comparison is made by the operator.

```python
# Not Equals Operator

if "Yes" != "No":
    # evaluates to True
    print("They are NOT equal")

val1 = 10
val2 = 20

if val1 != val2:
    print("They are NOT equal")

if (10 > 1) != (10 > 1000):
    # True != False
    print("They are NOT equal")
```

List Method `.count()`

The `.count()` Python list method searches a list for whatever search term it receives as an argument, then returns the number of matching entries found.

```python
backpack = ['pencil', 'pen', 'notebook',
'textbook', 'pen', 'highlighter', 'pen']
numPen = backpack.count('pen')

print(numPen)
# Output: 3
```

Adding Lists Together

In Python, lists can be added to each other using the plus symbol + . As shown in the code block, this will result in a new list containing the same items in the same order with the first list's items coming first.

Note: This will not work for adding one item at a time (use `.append()` method). In order to add one item, create a new list with a single value and then use the plus symbol to add the list.

```python
items = ['cake', 'cookie', 'bread']
total_items = items + ['biscuit', 'tart']
print(total_items)
# Result: ['cake', 'cookie', 'bread',
'biscuit', 'tart']
```

Determining List Length with `len()`

The Python `len()` function can be used to determine the number of items found in the list it accepts as an argument.

```python
knapsack = [2, 4, 3, 7, 10]
size = len(knapsack)
print(size)
# Output: 5
```

List Method `.append()`

In Python, you can add values to the end of a list using the `.append()` method. This will place the object passed in as a new element at the very end of the list. Printing the list afterwards will visually show the appended value. This `.append()` method is *not* to be confused with returning an entirely new list with the passed object.

```python
orders = ['daisies', 'periwinkle']
orders.append('tulips')
print(orders)
# Result: ['daisies', 'periwinkle', 'tulips']
```

List Indices

Python list elements are ordered by *index,* a number referring to their placement in the list. List indices start at 0 and increment by one.

To access a list element by index, square bracket notation is used: `list[index]` .

```python
berries = ["blueberry", "cranberry",
"raspberry"]

berries[0]    # "blueberry"
berries[2]    # "raspberry"
```

Negative List Indices

Negative indices for lists in Python can be used to reference elements in relation to the end of a list. This can be used to access single list elements or as part of defining a list range. For instance:

- To select the last element, my_list[-1] .
- To select the last three elements, my_list[-3:] .
- To select everything except the last two elements, my_list[:-2] .

```python
soups = ['minestrone', 'lentil', 'pho',
'laksa']
soups[-1]    # 'laksa'
soups[-3:]  # 'lentil', 'pho', 'laksa'
soups[:-2]  # 'minestrone', 'lentil'
```

sorted() Function

The Python sorted() function accepts a list as an argument, and will return a new, sorted list containing the same elements as the original. Numerical lists will be sorted in ascending order, and lists of Strings will be sorted into alphabetical order. It does not modify the original, unsorted list.

```python
unsortedList = [4, 2, 1, 3]
sortedList = sorted(unsortedList)
print(sortedList)
# Output: [1, 2, 3, 4]
```

Zero-Indexing

In Python, list index begins at zero and ends at the length of the list minus one. For example, in this list, 'Andy' is found at index 2 .

```python
names = ['Roger', 'Rafael', 'Andy', 'Novak']
```

List Slicing

A *slice*, or sub-list of Python list elements can be selected from a list using a colon-separated starting and ending point.
The syntax pattern is
myList[START_NUMBER:END_NUMBER] . The slice will include the START_NUMBER index, and everything until but excluding the END_NUMBER item.
When slicing a list, a new list is returned, so if the slice is saved and then altered, the original list remains the same.

```python
tools = ['pen', 'hammer', 'lever']
tools_slice = tools[1:3] # ['hammer',
'lever']
tools_slice[0] = 'nail'

# Original list is unaltered:
print(tools) # ['pen', 'hammer', 'lever']
```

Function Parameters

Sometimes functions require input to provide data for their code. This input is defined using *parameters*. *Parameters* are variables that are defined in the function definition. They are assigned the values which were passed as arguments when the function was called, elsewhere in the code.

```python
def write_a_book(character, setting,
special_skill):
  print(character + " is in " +
      setting + " practicing her " +
      special_skill)
```

For example, the function definition defines parameters for a character, a setting, and a skill, which are used as inputs to write the first sentence of a book.

Multiple Parameters

Python functions can have multiple *parameters*. Just as you wouldn't go to school without both a backpack and a pencil case, functions may also need more than one input to carry out their operations.
To define a function with multiple parameters, parameter names are placed one after another, separated by commas, within the parentheses of the function definition.

```python
def ready_for_school(backpack, pencil_case):
  if (backpack == 'full' and pencil_case ==
'full'):
    print ("I'm ready for school!")
```

Function Indentation

Python uses indentation to identify blocks of code. Code within the same block should be indented at the same level. A Python function is one type of code block. All code under a function declaration should be indented to identify it as part of the function. There can be additional indentation within a function to handle other statements such as for and if so long as the lines are not indented less than the first line of the function code.

```python
# Indentation is used to identify code blocks

def testfunction(number):
  # This code is part of testfunction
  print("Inside the testfunction")
  sum = 0
  for x in range(number):
    # More indentation because 'for' has a
code block
    # but still part of he function
    sum += x
  return sum
print("This is not part of testfunction")
```

Function Arguments

Parameters in python are variables — placeholders for the actual values the function needs. When the function is *called*, these values are passed in as *arguments*.
For example, the arguments passed into the function .sales() are the "The Farmer's Market", "toothpaste", and "$1" which correspond to the parameters grocery_store , item_on_sale , and cost .

```python
def sales(grocery_store, item_on_sale, cost):
  print(grocery_store + " is selling " +
item_on_sale + " for " + cost)

sales("The Farmer's Market", "toothpaste",
"$1")
```

Returning Value from Function

A return keyword is used to return a value from a Python function. The value returned from a function can

```python
def check_leap_year(year):
  if year % 4 == 0:
```

be assigned to a variable which can then be used in the program.

In the example, the function `check_leap_year` returns a string which indicates if the passed parameter is a leap year or not.

```python
    return str(year) + " is a
  else:
    return str(year) + " is not a leap year."


year_to_check = 2018
returned_value =
check_leap_year(year_to_check)
print(returned_value) # 2018 is not a leap
year.
```

Parameters as Local Variables

Function parameters behave identically to a function's local variables. They are initialized with the values passed into the function when it was called.

Like local variables, parameters cannot be referenced from outside the scope of the function.

In the example, the parameter `value` is defined as part of the definition of `my_function`, and therefore can only be accessed within `my_function`. Attempting to print the contents of `value` from outside the function causes an error.

```python
def my_function(value):
  print(value)


# Pass the value 7 into the function
my_function(7)


# Causes an error as `value` no longer exists
print(value)
```