CONTROL FLOW

If Statement

An `if` statement is a conditional statement that runs or skips code based on whether a condition is true or false. Here's a simple example.

```
if phone_balance < 5:
    phone_balance += 10
    bank_balance -= 10
```

Let's break this down.

1. An `if` statement starts with the `if` keyword, followed by the condition to be checked, in this case `phone_balance < 5`, and then a colon. The condition is specified in a boolean expression that evaluates to either True or False.

2. After this line is an indented block of code to be executed if that condition is true. Here, the lines that increment `phone_balance` and decrement `bank_balance` only execute if it is true that `phone_balance` is less than 5. If not, the code in this `if` block is simply skipped.

Use Comparison Operators in Conditional Statements

You have learned about Python's comparison operators (e.g. `==` and `!=`) and how they are different from assignment operators (e.g. `=`). In conditional statements, you want to use comparison operators. For example, you'd want to use `if x == 5` rather than `if x = 5`. If your conditional statement is causing a syntax error or doing something unexpected, check whether you have written `==` or `=`!

If, Elif, Else

In addition to the `if` clause, there are two other optional clauses often used with an `if` statement. For example:

```
if season == 'spring':
    print('plant the garden!')
elif season == 'summer':
    print('water the garden!')
elif season == 'fall':
    print('harvest the garden!')
elif season == 'winter':
    print('stay indoors!')
else:
    print('unrecognized season')
```

1. `if`: An `if` statement must always start with an `if` clause, which contains the first condition that is checked. If this evaluates to True, Python runs the code indented in this `if` block and then skips to the rest of the code after the `if` statement.

2. `elif`: `elif` is short for "else if." An `elif` clause is used to check for an additional condition if the conditions in the previous clauses in the `if` statement evaluate to False. As you can see in the example, you can have multiple `elif` blocks to handle different situations.

3. `else`: Last is the `else` clause, which must come at the end of an `if` statement if used. This clause doesn't require a condition. The code in an `else` block is run if all conditions above that in the `if`statement evaluate to False.

Examples

1. First Example - try changing the value of phone_balance

```
phone_balance = 10
bank_balance = 50

if phone_balance < 10:
   phone_balance += 10
   bank_balance -= 10

print(phone_balance)
print(bank_balance)
```

2. Second Example - try changing the value of number

```
number = 145
if number % 2 == 0:
   print("Number " + str(number) + " is even.")
else:
   print("Number " + str(number) + " is odd.")
```

3. Third Example - try to change the value of age

```
age = 35
 # Here are the age limits for bus fares
free_up_to_age = 4
child_up_to_age = 18
senior_from_age = 65
 # These lines determine the bus fare prices
concession_ticket = 1.25
adult_ticket = 2.50

 # Here is the logic for bus fare prices
if age <= free_up_to_age:
   ticket_price = 0
elif age <= child_up_to_age:
   ticket_price = concession_ticket
elif age >= senior_from_age:
   ticket_price = concession_ticket
else:
   ticket_price = adult_ticket

message = "Somebody who is {} years old will pay ${} to ride the bus.".format(age, ticket_price)
print(message)
```

Complex Boolean Expressions

`If` statements sometimes use more complicated boolean expressions for their conditions. They may contain multiple comparisons operators, logical operators, and even calculations. Examples:

```
if 18.5 <= weight / height**2 < 25:
    print("BMI is considered 'normal'")

if is_raining and is_sunny:
    print("Is there a rainbow?")

if (not unsubscribed) and (location == "USA" or location == "CAN"):
    print("send email")
```

For really complicated conditions you might need to combine some `and`s, `or`s and `not`s together. Use parentheses if you need to make the combinations clear.
However simple or complex, the condition in an `if` statement must be a boolean expression that evaluates to either True or False and it is this value that decides whether the indented block in an `if`statement executes or not.

Good and Bad Examples

Here are some things to keep in mind while writing boolean expressions for your `if` statements.

1. Don't use `True` or `False` as conditions

```
# Bad example
if True:
    print("This indented code will always get run.")
```

While "True" is a valid boolean expression, it's not useful as a condition since it always evaluates to True, so the indented code will always get run. Similarly, `if False` is not a condition you should use either - the statement following this `if` statement would never be executed.

```
# Another bad example
if is_cold or not is_cold:
    print("This indented code will always get run.")
```

Similarly, it's useless to use any condition that you know will always evaluate to True, like this example above. A boolean variable can only be True or False, so either `is_cold` or `not is_cold` is always True, and the indented code will always be run.

2. Be careful writing expressions that use logical operators

Logical operators `and`, `or` and `not` have specific meanings that aren't quite the same as their meanings in plain English. Make sure your boolean expressions are being evaluated the way you expect them to.

```
# Bad example
if weather == "snow" or "rain":
    print("Wear boots!")
```

This code is valid in Python, but it is not a boolean expression, although it reads like one. The reason is that the expression to the right of the `or` operator, `"rain"`, is not a boolean expression - it's a string! Later we'll discuss what happens when you use non-boolean-type objects in place of booleans.

3. Don't compare a boolean variable with `== True` or `== False`

This comparison isn't necessary, since the boolean variable itself is a boolean expression.

```
# Bad example
if is_cold == True:
    print("The weather is cold!")
```

This is a valid condition, but we can make the code more readable by using the variable itself as the condition instead, as below.

```
# Good example
if is_cold:
    print("The weather is cold!")
```

If you want to check whether a boolean is False, you can use the `not` operator.

Truth Value Testing

If we use a non-boolean object as a condition in an `if` statement in place of the boolean expression, Python will check for its truth value and use that to decide whether or not to run the indented code. By default, the truth value of an object in Python is considered True unless specified as False in the documentation.
Here are most of the built-in objects that are considered False in Python:

- constants defined to be false: `None` and `False`
- zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- empty sequences and collections: `""`, `()`, `[]`, `{}`, `set()`, `range(0)`

Example:

```
errors = 3
if errors:
    print("You have {} errors to fix!".format(errors))
else:
    print("No errors to fix!")
```

In this code, `errors` has the truth value True because it's a non-zero number, so the error message is printed. This is a nice, succinct way of writing an `if` statement.

`For` Loops

Python has two kinds of loops - `for` loops and `while` loops. A `for` loop is used to "iterate", or do something repeatedly, over an **iterable**.
An **iterable** is an object that can return one of its elements at a time. This can include sequence types, such as strings, lists, and tuples, as well as non-sequence types, such as dictionaries and files.

Example

Let's break down the components of a `for` loop, using this example with the list `cities`:

```
cities = ['new york city', 'mountain view', 'chicago', 'los angeles']
for city in cities:
    print(city)
print("Done!")
```

Components of a `For` Loop

1. The first line of the loop starts with the `for` keyword, which signals that this is a `for` loop
2. Following that is `city in cities`, indicating `city` is the iteration variable, and `cities` is the iterable being looped over. In the first iteration of the loop, `city` gets the value of the first element in `cities`, which is “new york city”.
3. The `for` loop heading line always ends with a colon `:`
4. Following the `for` loop heading is an indented block of code, the body of the loop, to be executed in each iteration of this loop. There is only one line in the body of this loop - `print(city)`.
5. After the body of the loop has executed, we don't move on to the next line yet; we go back to the `for` heading line, where the iteration variable takes the value of the next element of the iterable. In the second iteration of the loop above, `city` takes the value of the next element in `cities`, which is "mountain view".
6. This process repeats until the loop has iterated through all the elements of the iterable. Then, we move on to the line that follows the body of the loop - in this case, `print("Done!")`. We can tell what the next line after the body of the loop is because it is unindented. Here is another reason why paying attention to your indentation is very important in Python!
   Executing the code in the example above produces this output:

```
new york city
mountain view
chicago
```

```
los angeles
Done!
```

You can name iteration variables however you like. A common pattern is to give the iteration variable and iterable the same names, except the singular and plural versions respectively (e.g., 'city' and 'cities').

Using the `Range()` Function with `For` Loops

`range()` is a built-in function used to create an iterable sequence of numbers. You will frequently use `range()` with a `for` loop to repeat an action a certain number of times, as in this example:

```
for i in range(3):
    print("Hello!")
```

Output:

```
Hello!
Hello!
Hello!
```

`range(start=0, stop, step=1)`

The `range()` function takes three integer arguments, the first and third of which are optional:

- The 'start' argument is the first number of the sequence. If unspecified, 'start' defaults to 0.
- The 'stop' argument is 1 more than the last number of the sequence. This argument must be specified.
- The 'step' argument is the difference between each number in the sequence. If unspecified, 'step' defaults to 1.

Notes on using `range()`:

- If you specify one integer inside the parentheses with`range()`, it's used as the value for 'stop,' and the defaults are used for the other two.
  e.g. - `range(4)` returns `0, 1, 2, 3`
- If you specify two integers inside the parentheses with`range()`, they're used for 'start' and 'stop,' and the default is used for 'step.'
  e.g. - `range(2, 6)` returns `2, 3, 4, 5`
- Or you can specify all three integers for 'start', 'stop', and 'step.'
  e.g. - `range(1, 10, 2)` returns `1, 3, 5, 7, 9`

Creating and Modifying Lists

In addition to extracting information from lists, as we did in the first example above, you can also create and modify lists with `for` loops. You can **create** a list by appending to a new list at each iteration of the `for` loop like this:

```
# Creating a new list
cities = ['new york city', 'mountain view', 'chicago', 'los angeles']
capitalized_cities = []

for city in cities:
    capitalized_cities.append(city.title())
```

Modifying a list is a bit more involved, and requires the use of the `range()` function.
We can use the `range()` function to generate the indices for each value in the `cities` list. This lets us access the elements of the list with `cities[index]` so that we can modify the values in the `cities` list in place.

```
cities = ['new york city', 'mountain view', 'chicago', 'los angeles']

for index in range(len(cities)):
    cities[index] = cities[index].title()
```