Flow Control

1. Introduction

2. Elements of Flow Control

3. Importing Modules

Introduction

Last week, we learned the basics of individual instructions and that a program is just a series of instructions. But programming's real strength isn't just running one instruction after another!

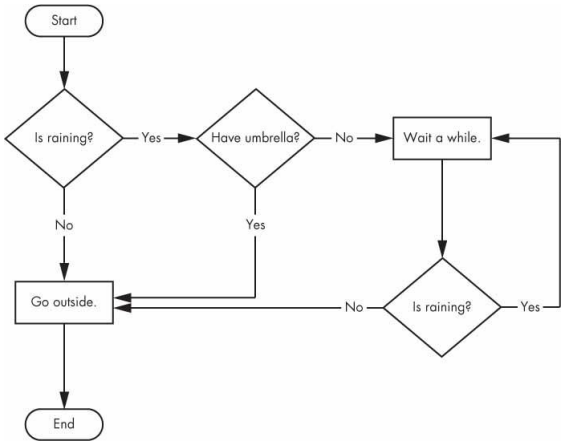
Last week, we learned the basics of individual instructions and that a program is just a series of instructions. But programming's real strength isn't just running one instruction after another!

A program can decide to skip instructions, repeat them, or choose one of several instructions to run! ***Flow control statements*** can decide which Python instructions to execute under which conditions.

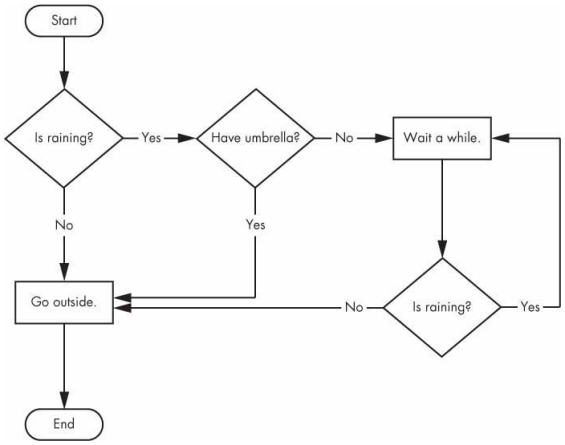
Last week, we learned the basics of individual instructions and that a program is just a series of instructions. But programming's real strength isn't just running one instruction after another!

A program can decide to skip instructions, repeat them, or choose one of several instructions to run! ***Flow control statements*** can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart!



source: https://automatetheboringstuff.com/2e/chapter2/



source: https://automatetheboringstuff.com/2e/chapter2/

But before you learn about flow control statements, you first need to learn how to represent those **yes** and **no** options and understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

Boolean expressions

A ***boolean expression*** is an expression that is either True or False. The following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

A ***boolean expression*** is an expression that is either True or False. The following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

In [2]:
```
print(5 == 5)
print(5 == 6)
```

```
True
False
```

A ***boolean expression*** is an expression that is either True or False. The following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

In [2]:
```
print(5 == 5)
print(5 == 6)
```

```
True
False
```

`True` and `False` are special values that belong to the class `bool` which are ***Boolean values***; they are not strings:

A ***boolean expression*** is an expression that is either True or False. The following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

In [2]:
```
print(5 == 5)
print(5 == 6)
```

```
True
False
```

`True` and `False` are special values that belong to the class `bool` which are ***Boolean values***; they are not strings:

In [3]:
```
type(True), type(False)
```

Out[3]:
```
(bool, bool)
```

The `==` operator is one of the ***comparison operators*** or relational operators; the others are:

The `==` operator is one of the ***comparison operators*** or relational operators; the others are:

	Meaning
x != y	x is not equal to y
x > y	x is greater than y
x < y	x is less than y
x >= y	x is greater than or equal to y
x <= y	x is less than or equal to y
x is y	x is the same as y
x is not y	x is not the same as y

These operators evaluate to `True` or `False` depending on the values you give them and, therefore, can be used in the decision point as a condition statement.

These operators evaluate to `True` or `False` depending on the values you give them and, therefore, can be used in the decision point as a condition statement.

In [4]:
```
print(42==42)
print(42==42.0)          # It will compare its value!
print(42=='42')          # int/float are always different from string
print(2!=3)
print('hello'=='Hello') # Python is case sensitive
print(42 < 100)
print(42 >= 100)
```

```
True
True
False
True
False
True
False
```

In [5]:
```
display_quiz(path+"bool.json", max_width=800)
```

Which of the following is a Boolean expression? Select all that apply.

3 + 4	3 + 4 == 7	3 == 4	True	"False"

Boolean (Logical) Operators

The three ***Boolean operators*** (`and`, `or`, and `not`) are used to operate on Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail.

The three ***Boolean operators*** (`and`, `or`, and `not`) are used to operate on Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail.

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

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

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

In [6]:
```
print((4 < 5) and (5 < 6))
print((6 < 5) or (9 < 6))
print((1 == 2) or (2 == 2))
print(not (1==3) and (3==4))
```

```
True
False
True
False
```

In [6]:
```
print((4 < 5) and (5 < 6))
print((6 < 5) or (9 < 6))
print((1 == 2) or (2 == 2))
print(not (1==3) and (3==4))
```

```
True
False
True
False
```

The computer will evaluate the left expression first, and then it will evaluate the right expression.

In [6]:
```
print((4 < 5) and (5 < 6))
print((6 < 5) or (9 < 6))
print((1 == 2) or (2 == 2))
print(not (1==3) and (3==4))
```

```
True
False
True
False
```

The computer will evaluate the left expression first, and then it will evaluate the right expression.

The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the `not` operators first, then the `and` operators, and then the `or` operators.

Arithmetic operators take precedence over logical operators. Python will always evaluate the arithmetic operators first. Next comes the relational operators. Finally, the logical operators are done last.

Arithmetic operators take precedence over logical operators. Python will always evaluate the arithmetic operators first. Next comes the relational operators. Finally, the logical operators are done last.

Level	Category	Operators
7 (high)	exponent	**
6	multiplication	*, /, //, %
5	addition	+, -
4	relational	==, !=, <=, >=, >, <
3	logical	not
2	logical	and
1 (low)	logical	or

In [7]:
```
display_quiz(path+"logical.json", max_width=800)
```

What is the correct Python expression for checking to see if a number stored in a variable x is between 0 and 5.

0 < x < 5	x > 0 and < 5	x > 0 or x < 5	x > 0 and x < 5

Elements of Flow Control

It can be shown that all programs could be written using three forms of control—namely, ***sequential execution, the selection statement and the repetition statement***. This is the idea behind ***structured programming***.

It can be shown that all programs could be written using three forms of control—namely, ***sequential execution, the selection statement and the repetition statement***. This is the idea behind ***structured programming***.

Flow control statements often start with a part called the ***condition*** and are always followed by a block of code called the ***clause*** or body.

It can be shown that all programs could be written using three forms of control—namely, ***sequential execution, the selection statement and the repetition statement***. This is the idea behind ***structured programming***.

Flow control statements often start with a part called the ***condition*** and are always followed by a block of code called the ***clause*** or body.

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; the condition is just a more specific name in the context of flow control statements!

Blocks of Code

Lines of Python code can be grouped together in ***blocks***. You can tell when a block begins and ends from the ***indentation*** of the lines of code. There are three rules for blocks.

1. Blocks begin when the indentation increases.

Lines of Python code can be grouped together in ***blocks***. You can tell when a block begins and ends from the ***indentation*** of the lines of code. There are three rules for blocks.

1. Blocks begin when the indentation increases.

2. Blocks can contain other blocks.
3. Blocks end when the indentation decreases to zero or to a containing block's indentation.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

In [8]:
```
name = 'Mary'
password = 'swordfish'
if name == 'Mary':
    print('Hello, Mary')
    if password == 'swordfish':
        print('Access granted.')
    else:
        print('Wrong password.')
```

```
Hello, Mary
Access granted.
```

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

In [8]:
```
name = 'Mary'
password = 'swordfish'
if name == 'Mary':
    print('Hello, Mary')
    if password == 'swordfish':
        print('Access granted.')
    else:
        print('Wrong password.')
```

```
Hello, Mary
Access granted.
```

You can view the execution of this program at https://autbor.com/blocks/. The first block of code starts at the line `print('Hello, Mary')` and contains all the lines after it. Inside this block is another block, which has only a single line in it: `print('Access Granted.')`. The third block is also one line long: `print('Wrong password.')`.

An `IndentationError` occurs if you have more than one statement in a block and those statements do not have the same indentation:

An `IndentationError` occurs if you have more than one statement in a block and those statements do not have the same indentation:

In [9]:
```
name = 'Mary'
password = 'swordfish'
if name == 'Mary':
  print('Hello, Mary')
    if password == 'swordfish':
        print('Access granted.')
    else:
        print('Wrong password.')
```

```
  File "C:\Users\adm\AppData\Local\Temp\ipykernel_35736\140440185.py",
line 5
    if password == 'swordfish':
    ^
IndentationError: unexpected indent
```

In [10]:
```
display_quiz(path+"block.json", max_width=800)
```

How many lines of code (statement) can appear in the indented code block below the if and else lines?

- Zero or more.
- One or more, and each must contain the same number.
- One or more.
- Just one.

Conditional execution

The control statement affords us a mechanism for jumping from one part of a program to another. This enables what is called ***control structures***.

The control statement affords us a mechanism for jumping from one part of a program to another. This enables what is called ***control structures***.

One example of this is the ***if-statement***. An `if` statement's body (that is, the block following the `if` statement) will execute if the statement's condition is `True`. The body is skipped if the condition is `False`.

The control statement affords us a mechanism for jumping from one part of a program to another. This enables what is called ***control structures***.

One example of this is the ***if-statement***. An `if` statement's body (that is, the block following the `if` statement) will execute if the statement's condition is `True`. The body is skipped if the condition is `False`.

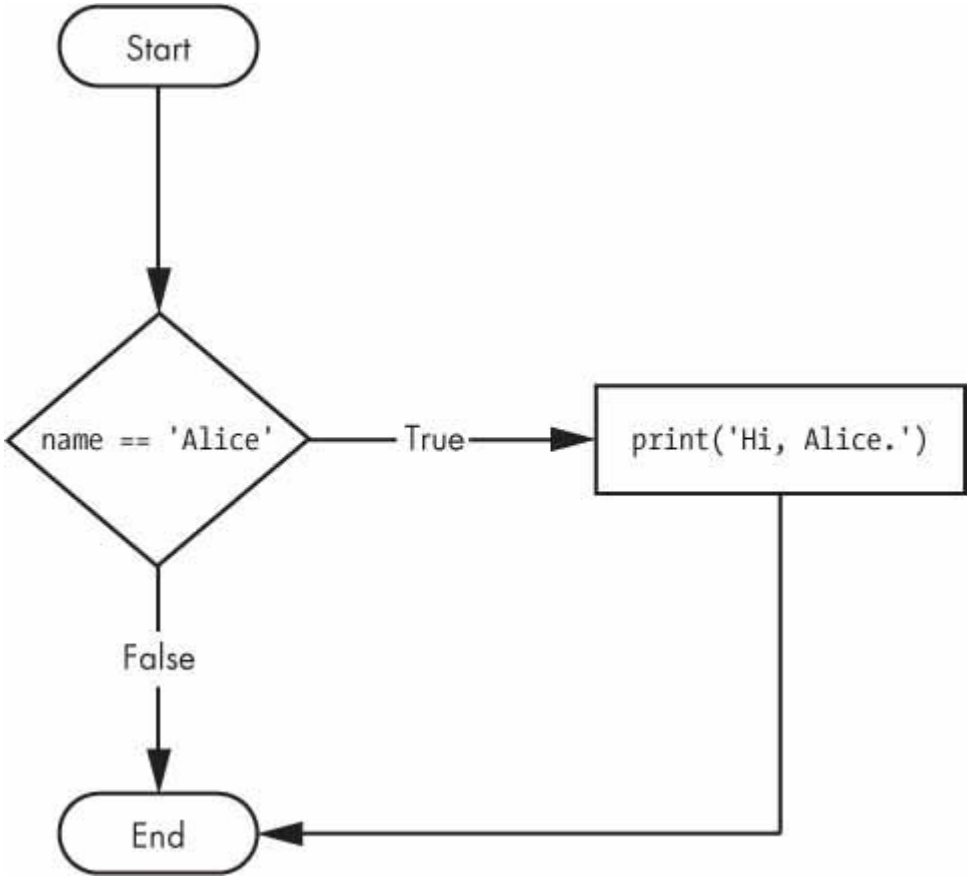
In Python, an `if` statement consists of the following:

- The `if` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `if` body)

The boolean expression after the `if` statement is called the condition. We end the `if` statement with a colon character (`:`) and the line(s) after the `if` statement are indented. If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

The boolean expression after the `if` statement is called the condition. We end the `if` statement with a colon character (`:`) and the line(s) after the `if` statement are indented. If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

In [ ]:
```
name = 'Mary'
if name == 'Alice':
    print('Hi, Alice.')
```



source: https://automatetheboringstuff.com/2e/chapter2/

A second form of the `if` statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

A second form of the `if` statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

In [11]:
```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

```
Hello, stranger.
```

A second form of the `if` statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

In [11]:
```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

```
Hello, stranger.
```

You can also write the above code in one line using the ***ternary conditional operator***:

A second form of the `if` statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

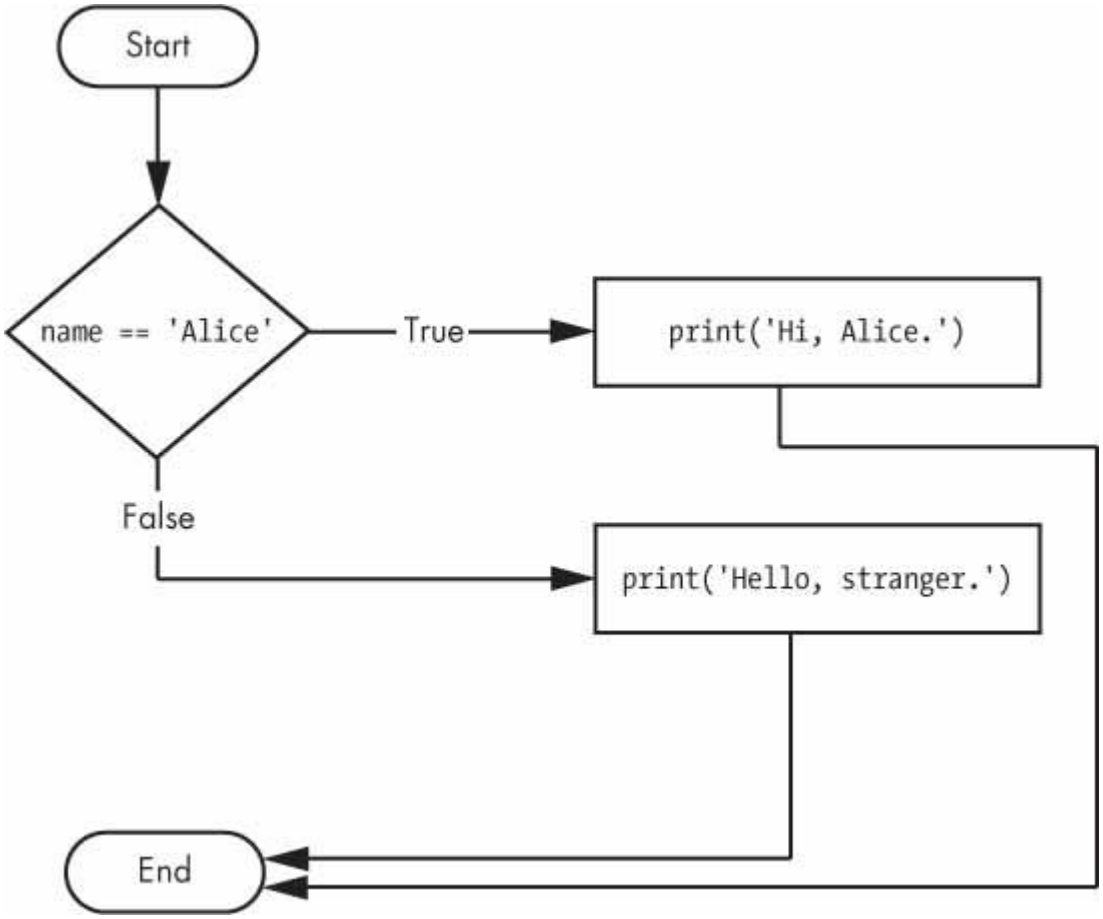
In [11]:
```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

```
Hello, stranger.
```

You can also write the above code in one line using the ***ternary conditional operator***:

In [12]:
```
print('Hi, Alice.') if name == 'Alice' else print('Hello, stranger.') # Note
```

```
Hello, stranger.
```



source: https://automatetheboringstuff.com/2e/chapter2/

Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

In [13]:
```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kidd.')
else:
    print('You are neither Alice nor a little kid.')
```

```
You are neither Alice nor a little kid.
```

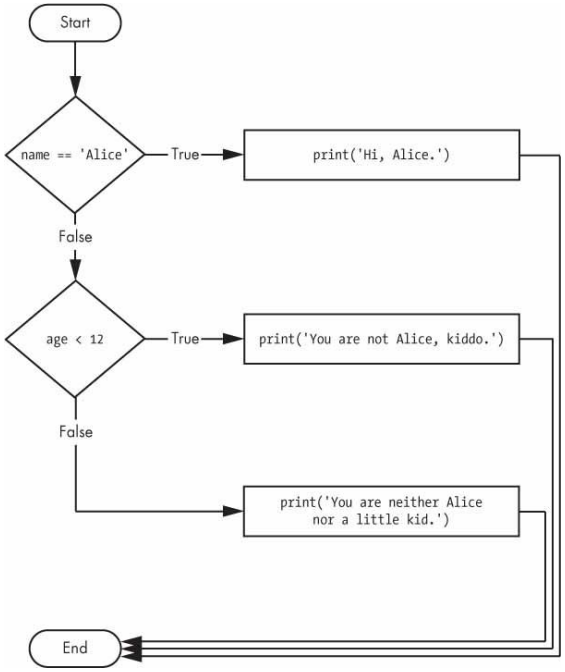
Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

In [13]:
```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kidd.')
else:
    print('You are neither Alice nor a little kid.')
```

```
You are neither Alice nor a little kid.
```

You can view the execution of this program at https://autbor.com/littlekid/. In plain English, this type of flow control structure would be “If the first condition is true, do this. Else, if the second condition is true, do that.



source: https://automatetheboringstuff.com/2e/chapter2/

In [14]:
```
display_quiz(path+"conditions.json", max_width=800)
```

What will the following code print if x = 3, y = 5, and z = 2?

```
if x < y and x < z:
    print("a")
elif y < x and y < z:
    print("b")
else:
    print("c")
```

b

a

c

Exercise 1: Write a program that prompts the user to enter a `row` and `column` (each between 0 and 7) corresponding to an chessboard square, then prints ”black” or ”white” depending on the square’s color; if either input is outside the 0–7 range, it prints ”out of board.”

> Exercise 1: Write a program that prompts the user to enter a `row` and `column` (each between 0 and 7) corresponding to an chessboard square, then prints "black" or "white" depending on the square's color; if either input is outside the 0–7 range, it prints "out of board."

source: https://inventwithpython.com/pythongently/images/image011.png

In [ ]:

```
row = int(input("Enter row :"))
column = int(input("Enter column :"))
# If the column and row is out of bounds, print out of board:
if column ____ or column ___ or row ___ or row ____:
    print('out of board')
# If the even/oddness of the column and row match, print 'white':
____ column % _ == row % _:
    print('white')
# If they don't match, then print 'black':
____:
    print('black')
```

Loops and Iterations

You can make a block of code execute over and over again using a ***`while` statement***. The code in a `while` body will be executed as long as the `while` statement's condition is `True`. A `while` statement always consists of the following:

You can make a block of code execute over and over again using a ***`while` statement***. The code in a `while` body will be executed as long as the `while` statement's condition is `True`. A `while` statement always consists of the following:

- The `while` keyword
- A condition and a colon
- Starting on the next line, an indented block of code (called the `while` body)

You can make a block of code execute over and over again using a ***`while` statement***. The code in a `while` body will be executed as long as the `while` statement's condition is `True`. A `while` statement always consists of the following:

- The `while` keyword
- A condition and a colon
- Starting on the next line, an indented block of code (called the `while` body)

In [15]:
```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam += 1 # equivalent to spam = spam + 1
```

```
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
```

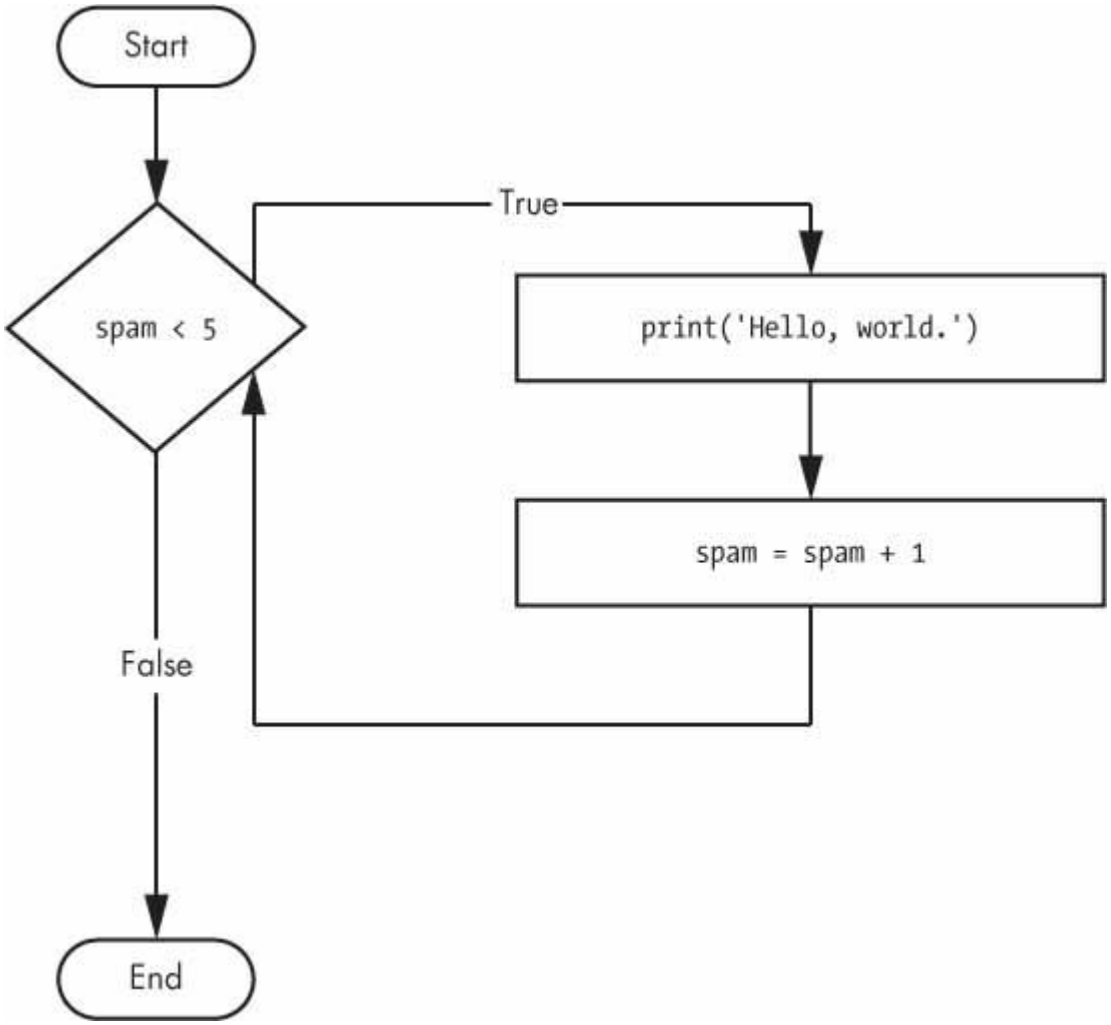
You can make a block of code execute over and over again using a ***`while` statement***. The code in a `while` body will be executed as long as the `while` statement's condition is `True`. A `while` statement always consists of the following:

- The `while` keyword
- A condition and a colon
- Starting on the next line, an indented block of code (called the `while` body)

In [15]:
```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam += 1 # equivalent to spam = spam + 1
```

```
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
```

Augmented assignments abbreviate assignment expressions in which the same variable name appears on the left and right of the assignment's `=` as above



source: https://automatetheboringstuff.com/2e/chapter2/

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `True` or `False`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `True` or `False`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

In the `while` ***loop***, the condition is always checked at the start of each ***iteration*** (that is, each time the loop is executed). If the condition is `True`, then the body is executed, and afterward, the condition is checked again. The first time the condition is found to be `False`, the while body is skipped.

A common programming pattern is that we can run the program as long as the user wants by putting most of the program in a `while` loop. We'll define a quit value to decide when to leave:

A common programming pattern is that we can run the program as long as the user wants by putting most of the program in a `while` loop. We'll define a quit value to decide when to leave:

In [16]:
```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "
message = ""
while message != 'quit':
    message = input(prompt)
    print(message)
```

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. hi
hi

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

A common programming pattern is that we can run the program as long as the user wants by putting most of the program in a `while` loop. We'll define a quit value to decide when to leave:

In [16]:
```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "
message = ""
while message != 'quit':
    message = input(prompt)
    print(message)
```

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. hi
hi

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

We first set up a variable `message` to keep track of whatever value the user enters. We define `message` as an empty string, `""`, so Python has something to check at the first time.

Note that Python considers `0`, `None`, empty string, and empty container as `False` and all other things are `True`!

Note that Python considers `0`, `None`, empty string, and empty container as `False` and all other things are `True`!

In [17]:
```
bool(""), bool(0), bool(None), bool(prompt), bool(12)
```

Out[17]:
```
(False, False, False, True, True)
```

Using `break` to Exit a Loop

The above program works well, except that it prints the word 'quit' as if it were an actual message. In fact, there is a shortcut to getting the program execution to break out of a `while` loop's body early. If the execution reaches a ***`break` statement***, it immediately exits the while loop's body！

The above program works well, except that it prints the word 'quit' as if it were an actual message. In fact, there is a shortcut to getting the program execution to break out of a `while` loop's body early. If the execution reaches a ***`break` statement***, it immediately exits the while loop's body !

In [19]:
```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "
message = ""
while True:
    message = input(prompt)
    if message == 'quit':
        break
    else:
        print(message)
```

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. hi
hi

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

The above program works well, except that it prints the word 'quit' as if it were an actual message. In fact, there is a shortcut to getting the program execution to break out of a `while` loop's body early. If the execution reaches a ***`break` statement***, it immediately exits the while loop's body !

In [19]:
```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "
message = ""
while True:
    message = input(prompt)
    if message == 'quit':
        break
    else:
        print(message)
```

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. hi
hi

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

The fourth line creates an ***infinite loop***; it is a `while` loop whose condition is always `True`. After the program execution enters this loop, it will exit the loop only when a `break` statement is executed.

`continue` Statemet

Rather than breaking out of a loop entirely without executing the rest of its code, you can use the ***`continue` statement*** to return to the beginning of the loop based on the result of a conditional test. For example, consider a loop that counts from 1 to 10 but prints only the odd numbers in that range:

Rather than breaking out of a loop entirely without executing the rest of its code, you can use the ***`continue` statement*** to return to the beginning of the loop based on the result of a conditional test. For example, consider a loop that counts from 1 to 10 but prints only the odd numbers in that range:

In [20]:
```
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue
    else:
        print(current_number, end=' ')
```

```
1 3 5 7 9
```

Rather than breaking out of a loop entirely without executing the rest of its code, you can use the ***`continue` statement*** to return to the beginning of the loop based on the result of a conditional test. For example, consider a loop that counts from 1 to 10 but prints only the odd numbers in that range:

In [20]:
```
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue
    else:
        print(current_number, end=' ')
```

```
1 3 5 7 9
```

Note that the built-in function `print()` displays its argument(s), then moves the cursor to the next line. You can change this behavior with the argument `end`. We used one space (' '), so each call to print displays the character's value followed by one space!

In [21]:
```
display_quiz(path+"while.json", max_width=800)
```

The following code contains an infinite loop. Which is the best explanation for why the loop does not terminate?

- You cannot compare n to 0 in while loop. You must compare it to another variable.
- In the while loop body, we must set n to False, and this code does not do that.
- answer starts at 1 and is incremented by n each time, so it will always be positive
- n starts at 10 and is incremented by 1 each time through the loop, so it will always be positive

"TRUTHY" and "FALSY" Values

Let us delve into the following program:

Let us delve into the following program:

In [22]:
```
name = ''
while not name:
    print('Enter your name:')
    name = input()

print('How many guests will you have?')
numOfGuests = int(input())

if numOfGuests:
    print('Be sure to have enough room for all your guests.')
print('Done')
```

```
Enter your name:
phonchi
How many guests will you have?
3
Be sure to have enough room for all your guests.
Done
```

Let us delve into the following program:

In [22]:
```
name = ''
while not name:
    print('Enter your name:')
    name = input()

print('How many guests will you have?')
numOfGuests = int(input())

if numOfGuests:
    print('Be sure to have enough room for all your guests.')
print('Done')
```

```
Enter your name:
phonchi
How many guests will you have?
3
Be sure to have enough room for all your guests.
Done
```

You can view the execution of this program at https://autbor.com/howmanyguests/.

`for` Loops and the `range()` Function

The `while` loop keeps looping `while` when its condition is `True`, but what if you want to execute a block of code **only a certain number of times**? You can do this with a ***`for` loop statement*** and the `range()` function.

The `while` loop keeps looping `while` when its condition is `True`, but what if you want to execute a block of code **only a certain number of times**? You can do this with a ***`for` loop statement*** and the `range()` function.

A `for` statement looks something like `for i in range(5)`: and includes the following:

- The `for` keyword
- A variable name
- The ***`in` keyword***

The `while` loop keeps looping `while` when its condition is `True`, but what if you want to execute a block of code **only a certain number of times**? You can do this with a ***`for` loop statement*** and the `range()` function.

A `for` statement looks something like `for i in range(5)`: and includes the following:

- The `for` keyword
- A variable name
- The ***`in` keyword***

- A call to the `range()` funtion with up to three integers passed to it (The `for` statement can iterate over a ***sequence*** item by item!)
- A colon
- Starting on the next line, an indented block of code (called the for body)

Let’s create a new program to help you see a `for` loop in action.

Let's create a new program to help you see a `for` loop in action.

In [23]:
```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

Let's create a new program to help you see a `for` loop in action.

In [23]:
```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

You can view the execution of this program at https://autbor.com/fivetimesfor/.

You can actually use a `while` loop to do the same thing as a `for` loop; `for` loops are just more concise.

You can actually use a `while` loop to do the same thing as a `for` loop; `for` loops are just more concise.

In [24]:
```
print('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

The Starting, Stopping, and Stepping Arguments to `range()`

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence of integers, including starting at a number other than zero.

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence of integers, including starting at a number other than zero.

In [25]:
```
for i in range(12, 16):
    print(i)
```

```
12
13
14
15
```

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence of integers, including starting at a number other than zero.

In [25]:
```
for i in range(12, 16):
    print(i)
```

```
12
13
14
15
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the step argument.

In [26]:
```
for i in range(0, 10, 2):
    print(i)
```

```
0
2
4
6
8
```

In [26]:
```
for i in range(0, 10, 2):
    print(i)
```

```
0
2
4
6
8
```

You can even use a **negative number** for the step argument to make the `for` loop count down instead of up.

In [26]:
```
for i in range(0, 10, 2):
    print(i)
```

```
0
2
4
6
8
```

You can even use a **negative number** for the step argument to make the `for` loop count down instead of up.

In [27]:
```
for i in range(5, -1, -1):
    print(i)
```

```
5
4
3
2
1
0
```

In [28]:
```
display_quiz(path+"for.json", max_width=800)
```

How many times is the word HELLO printed by the following statements?

```
s = "python rocks"
for ch in s:
    print("HELLO")
```

12	Error, the for statement needs to use the range() function.
10	11

> Exercise 2: Write a script that displays the following triangle patterns. Use `for` loops to generate the patterns.

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Hint: Try to use nested loops and use the outer loop to display each row while the inner loop to display each column

In [ ]:

```
for row in range(__,__):
    for column in range(__,__):
        print('*', end='')
    print()
```

Importing Modules

All Python programs can call a basic set of functions called **built-in functions**, including the `print()`, `input()`, `len()` and `range()` functions you've seen before.

All Python programs can call a basic set of functions called **built-in functions**, including the `print()`, `input()`, `len()` and `range()` functions you've seen before.

Python also comes with a set of modules called the ***standard library***. Each module is a Python program that contains a related group of functions that can be embedded in your programs.

All Python programs can call a basic set of functions called **built-in functions**, including the `print()`, `input()`, `len()` and `range()` functions you've seen before.

Python also comes with a set of modules called the ***standard library***. Each module is a Python program that contains a related group of functions that can be embedded in your programs.

For example, the `math` module has mathematics-related functions. The `random` module has random number-related functions, and so on.

Before you can use the functions in a module, you must ***import*** the module with an `import` statement. In code, an `import` statement consists of the following:

- The `import` keyword

Before you can use the functions in a module, you must ***import*** the module with an `import` statement. In code, an `import` statement consists of the following:

- The `import` keyword
- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

In [29]:
```
import random
for i in range(5):
    print(random.randint(1, 10))
```

```
3
8
2
1
7
```

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

In [29]:
```
import random
for i in range(5):
    print(random.randint(1, 10))
```

```
3
8
2
1
7
```

You can view the execution of this program at https://autbor.com/printrandom/. The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it.

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

In [29]:
```
import random
for i in range(5):
    print(random.randint(1, 10))
```

```
3
8
2
1
7
```

You can view the execution of this program at https://autbor.com/printrandom/. The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it.

Since `randint()` is in the `random` module, you must first type `random.` in front of the function name to tell Python to look for this function inside the `random` module.

Ending a Program Early with the `sys.exit()` Function

The last flow control concept to cover is how to terminate the program. Programs always terminate if the program execution reaches the bottom of the instructions.

The last flow control concept to cover is how to terminate the program. Programs always terminate if the program execution reaches the bottom of the instructions.

However, you can cause the program to terminate before the last instruction by calling the `sys.exit()`. Since this function is in the `sys` module, you have to `import sys` before your program can use it.

The last flow control concept to cover is how to terminate the program. Programs always terminate if the program execution reaches the bottom of the instructions.

However, you can cause the program to terminate before the last instruction by calling the `sys.exit()`. Since this function is in the `sys` module, you have to `import sys` before your program can use it.

In [30]:
```
%%writefile exit.py

import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
print('This line will not be printed')
```

```
Overwriting exit.py
```

In [31]:
```
%run exit.py
```

```
Type exit to exit.
hi
You typed hi.
Type exit to exit.
exit
```

In [31]:
```
%run exit.py
```

```
Type exit to exit.
hi
You typed hi.
Type exit to exit.
exit
```

By using expressions that evaluate to `True` or `False` (also called conditions), you can write programs that make decisions on what code to execute and what code to skip. You can also execute code over and over again in a loop while a certain condition evaluates to `True`.

In [31]:
```
%run exit.py
```

```
Type exit to exit.
hi
You typed hi.
Type exit to exit.
exit
```

By using expressions that evaluate to `True` or `False` (also called conditions), you can write programs that make decisions on what code to execute and what code to skip. You can also execute code over and over again in a loop while a certain condition evaluates to `True`.

These flow control statements will let you write more intelligent programs. You can also use another type of flow control by **writing your own functions**, which is the topic of the next chapter.

In [32]:
```
from jupytercards import display_flashcards
fpath= "https://raw.githubusercontent.com/phonchi/nsysu-math106A/refs/heads/m
display_flashcards(fpath + 'ch2.json')
```

Flow control statements

Next

>