More Looping Structures

- **while** loop
- **continue** – restart loop from the top
- **break** – jump out of a loop

Why do we need a while loop?

- We have a for loop that does these things very well:
  - iterate over *each* element of a sequence

```
for elem in sequence:
    # do some stuff
```

  - loop a *fixed number* of times

```
for i in range(num):
    # do some stuff
```

- So why do we need another loop?

A `for` loop may not work well because

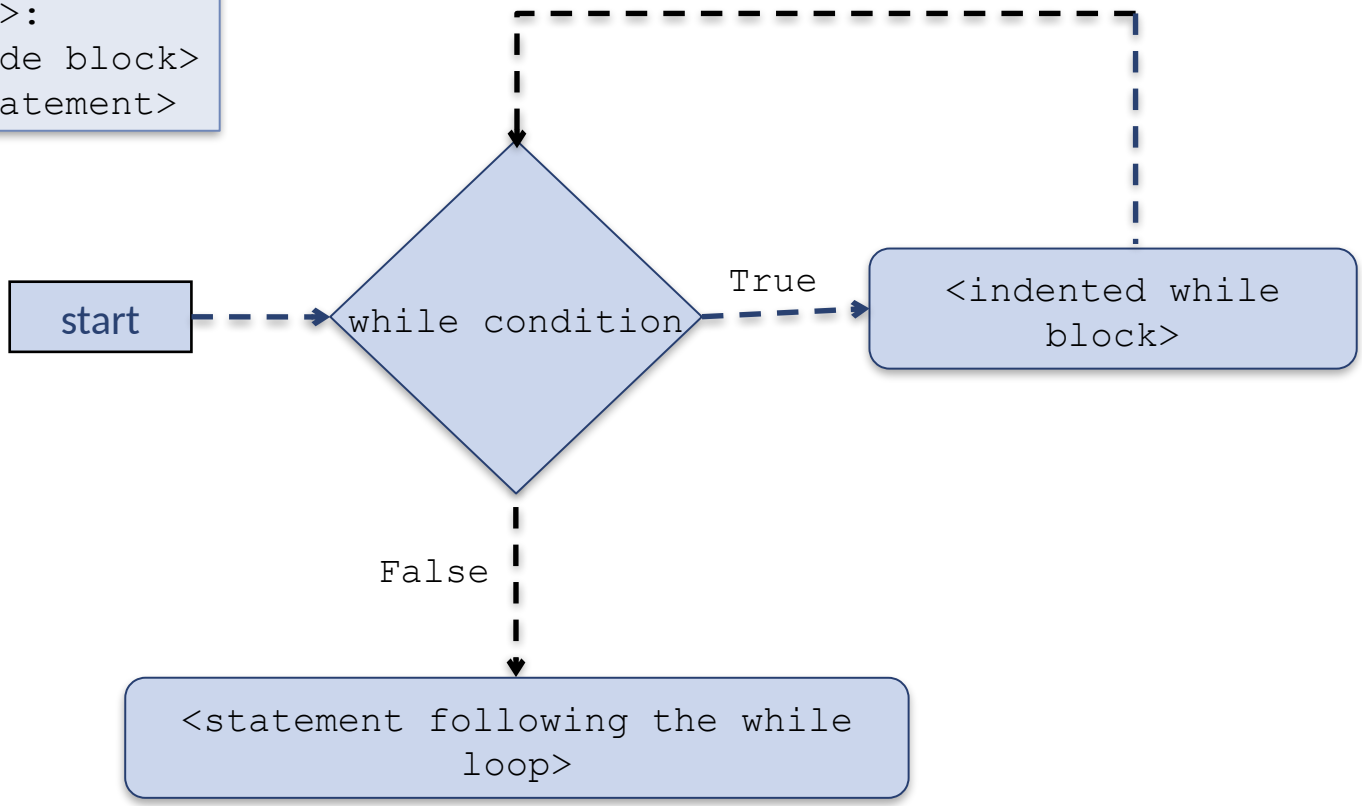
- The repetition may not be based on a iterable or sequential object
- We don't always know *how many times* to loop in advance
  - so

    ```
    for i in range(num):
    ```

    doesn't work
- Sometimes we need to refer to elements of a sequence that we have already iterated past or have not yet gotten to
  - so looping over elements one at a time isn't a natural way to repeat

A `while` loop lets you specify any controlling condition that you want

```
while <condition>:
    <indented code block>
<non-indented statement>
```



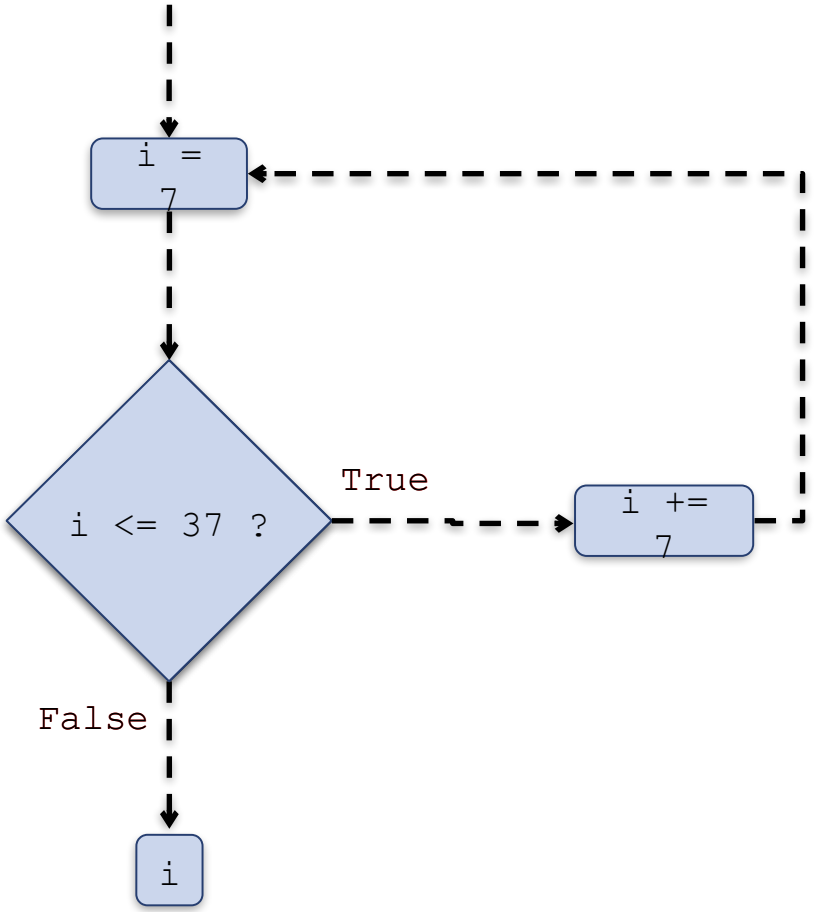
while loop

Example: compute the smallest multiple of 7 greater than 37.

Idea: generate multiples of 7 until we get a number greater than 37

```
>>> i = 7
>>> while i <= 37:
        i += 7

>>> i
42
```



Exercise

Write function `getNegativeNumber()` that uses a while loop to:

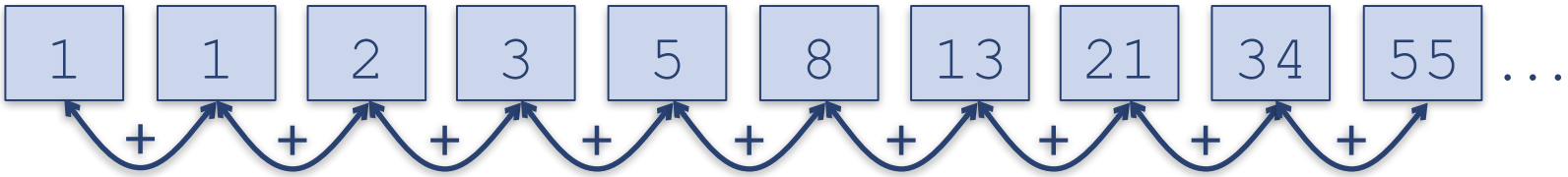
- prompt a user for input as many times as necessary to get a negative number
- return the negative number

Should the return statement be inside the while loop or after it? Why?

Sequence loop pattern

Generating a sequence that reaches the desired solution

Fibonacci sequence



Find the first Fibonacci number greater than some bound

```
def fibonacci(bound):
    '''loop through the Fibonacci numbers while the current
    Fibonacci number is less than or equal to bound,
    return the smallest Fibonacci number greater than bound'''

    # initialize the previous and current Fibonacci numbers
    previous = current = 1
    while current <= bound:
        # update the values of previous and current
        previous, current = current, previous+current
    return current
```

Infinite loop pattern

An infinite loop provides a continuous service

```
>>> hello()
What is your name? Sam
Hello Sam
What is your name? Tim
Hello Tim
What is your name? Alex
Hello Alex
What is your name?
```

This is an example of a greeting service. The server could be a time server, or a web server, or a mail server, or ...

```
def hello():
    '''provide a greeting service that repeatedly requests
       a user name and then greets the user'''

    while True:
        name = input('What is your name? ')
        print('Hello {}'.format(name))
```

Using a 'flag' to terminate a loop

Example: a function that creates a list of cities entered by the user and returns it

The empty string is a “flag” that indicates the end of the input

```
def cities():
    lst = []

    while True:
        city = input('Enter city: ')
        # if user enters empty string
        if city == '':
            return lst
        # else append city to lst
        lst.append(city)
```

```
>>> cities()
Enter city: Lisbon
Enter city: San Francisco
Enter city: Hong Kong
Enter city:
['Lisbon', 'San Francisco', 'Hong
Kong']
>>>
```

The break statement

The break statement:

- is used inside the body of a for or while loop
- terminates execution of the loop
- transfers execution to the statement that *follows* the loop

```
while True:
    city = input('Enter city: ')
        if city == '':
            return lst
        lst.append(city)
```

```
def cities2():
    lst = []
    while True:
        city = input('Enter city: ')
        if city == '':
            break
        lst.append(city)
    return lst
```

The continue statement

The continue statement:

- is used inside the body of a for or while loop
- interrupts the current iteration of the loop
- transfers execution to the top (for or while line) of the loop

```
def getAges(letter):
    ''' Get the age of every person whose name begins with letter'''
    rtnLst = []

    while True:
        name = input('Enter name: ')
        if name == '':
            return rtnLst
        if name[0] != letter:
            continue
        age = int(input('Enter age: '))
        rtnLst.append(age)
```

break / continue in nested loops

The continue/break statements:

- are used inside the body of a for or while loop
- when executed, they alter the flow of execution

For both continue and break, if the current loop is nested inside another loop, only the innermost loop is affected

```
def before(table):
    for row in table:
        for num in row:
            if num == 0:
                break
            print(num, end=' ')
        print()
```

```
def ignore(table):
    for row in table:
        for num in row:
            if num == 0:
                continue
            print(num, end=' ')
        print()
```

```
>>> table = [
        [2, 3, 0, 6],
        [0, 3, 4, 5],
        [4, 5, 6, 0]]
```

```
>>> before(table)
2 3

4 5 6
```

```
>>> ignore(table)
2 3 6
3 4 5
4 5 6
```