Imperative programming with Python

Class #3

Facundo Carreiro

ILLC, University of Amsterdam

January 2015

Functions: execution and the call stack

- ```
  def cat_twice_and_print(part1, part2):
      cat = part1 + part2
      print_twice(cat)

  def print_twice(msg):
      print msg
      print msg

  line1 = 'welcome␣'
  line2 = 'to␣the␣jungle'
  cat_twice_and_print(line1, line2)
  ```

__main__

Output:
  ```

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

__main__

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

__main__

Output:

Functions: execution and the call stack

- ```
  def cat_twice_and_print(part1, part2):
      cat = part1 + part2
      print_twice(cat)

  def print_twice(msg):
      print msg
      print msg

  line1 = 'welcome␣'
  line2 = 'to␣the␣jungle'
  cat_twice_and_print(line1, line2)
  ```

__main__

Output:
  ```

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

__main__

Output:

Functions: execution and the call stack

```
  def cat_twice_and_print(part1, part2):
      cat = part1 + part2
      print_twice(cat)

  def print_twice(msg):
▶     print msg
      print msg

  line1 = 'welcome␣'
  line2 = 'to␣the␣jungle'
  cat_twice_and_print(line1, line2)
```

__main__

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

__main__

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

__main__

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

__main__

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
► line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

__main__

line1 ↦ 'welcome '

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
▶ cat_twice_and_print(line1, line2)
```

__main__
line1 ↦ 'welcome '
line2 ↦ 'to the jungle'

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

cat_twice_and_print
part1 ↦ 'welcome '
part2 ↦ 'to the jungle'

__main__
line1 ↦ 'welcome '
line2 ↦ 'to the jungle'

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

cat_twice_and_print
part1 ↦ 'welcome '
part2 ↦ 'to the jungle'

__main__
line1 ↦ 'welcome '
line2 ↦ 'to the jungle'

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

cat_twice_and_print
part1 ↦ 'welcome '
part2 ↦ 'to the jungle'
cat ↦ 'welcome to the jungle'

__main__
line1 ↦ 'welcome '
line2 ↦ 'to the jungle'

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

print_twice
msg ↦ 'welcome to the jungle'

cat_twice_and_print
part1 ↦ 'welcome '
part2 ↦ 'to the jungle'
cat ↦ 'welcome to the jungle'

__main__
line1 ↦ 'welcome '
line2 ↦ 'to the jungle'

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

print_twice
msg $\mapsto$ 'welcome to the jungle'

cat_twice_and_print
part1 $\mapsto$ 'welcome '
part2 $\mapsto$ 'to the jungle'
cat $\mapsto$ 'welcome to the jungle'

__main__
line1 $\mapsto$ 'welcome '
line2 $\mapsto$ 'to the jungle'

Output:

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
►   print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

print_twice
msg ↦ 'welcome to the jungle'

cat_twice_and_print
part1 ↦ 'welcome '
part2 ↦ 'to the jungle'
cat ↦ 'welcome to the jungle'

__main__
line1 ↦ 'welcome '
line2 ↦ 'to the jungle'

Output:

```
welcome to the jungle
```

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

cat_twice_and_print
part1 $\mapsto$ 'welcome '
part2 $\mapsto$ 'to the jungle'
cat $\mapsto$ 'welcome to the jungle'

__main__
line1 $\mapsto$ 'welcome '
line2 $\mapsto$ 'to the jungle'

Output:

```
welcome to the jungle
welcome to the jungle
```

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
► cat_twice_and_print(line1, line2)
```

__main__
line1 ↦ 'welcome '
line2 ↦ 'to the jungle'

Output:

```
welcome to the jungle
welcome to the jungle
```

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    f(msg)
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

print_twice
msg ↦ 'welcome to the jungle'

cat_twice_and_print
part1 ↦ 'welcome '
part2 ↦ 'to the jungle'
cat ↦ 'welcome to the jungle'

__main__
line1 ↦ 'welcome '
line2 ↦ 'to the jungle'

Functions: execution and the call stack

```
def cat_twice_and_print(part1, part2):
    cat = part1 + part2
    print_twice(cat)

def print_twice(msg):
    f(msg)
    print msg
    print msg

line1 = 'welcome␣'
line2 = 'to␣the␣jungle'
cat_twice_and_print(line1, line2)
```

print_twice
msg ↦ 'welcome to the jungle'

cat_twice_and_print
part1 ↦ 'welcome '
part2 ↦ 'to the jungle'
cat ↦ 'welcome to the jungle'

__main__
line1 ↦ 'welcome '
line2 ↦ 'to the jungle'

```
Traceback (most recent call last):
  File "code.py", line 12, in <module>
    cat_twice_and_print(line1, line2)
  File "code.py", line 3, in cat_twice_and_print
    print_twice(cat)
  File "code.py", line 6, in print_twice
    f(msg)
NameError: global name 'f' is not defined
```

Functions: recursion

- Functions can call *themselves* in their definition.

```
# calculates n * m (in a complicated way)
def multiply(n, m):
    if n == 0:
        return 0
    else:
        return m + multiply(n - 1, m)
```

Functions: recursion

- Functions can call *themselves* in their definition.

```
# calculates n * m (in a complicated way)
def multiply(n, m):
    if n == 0:
        return 0
    else:
        return m + multiply(n - 1, m)
```

- How does the call stack look for `multiply(2, 7)` look like?

Functions: recursion

- Functions can call *themselves* in their definition.

```
# calculates n * m (in a complicated way)
def multiply(n, m):
    if n == 0:
        return 0
    else:
        return m + multiply(n - 1, m)
```

- How does the call stack look for `multiply(2, 7)` look like?

multiply `n` $\mapsto$ 2, `m` $\mapsto$ 7 `ret` $\mapsto 7 + \ldots$

Functions: recursion

- Functions can call *themselves* in their definition.

```
# calculates n * m (in a complicated way)
def multiply(n, m):
    if n == 0:
        return 0
    else:
        return m + multiply(n - 1, m)
```

- How does the call stack look for `multiply(2, 7)` look like?

multiply n $\mapsto$ 2, m $\mapsto$ 7 ret $\mapsto$ 7 + ...
multiply n $\mapsto$ 1, m $\mapsto$ 7 ret $\mapsto$ 7 + ...

Functions: recursion

- Functions can call *themselves* in their definition.

```
# calculates n * m (in a complicated way)
def multiply(n, m):
    if n == 0:
        return 0
    else:
        return m + multiply(n - 1, m)
```

- How does the call stack look for `multiply(2, 7)` look like?

multiply `n` $\mapsto$ 2, `m` $\mapsto$ 7 `ret` $\mapsto 7 + \ldots$
multiply `n` $\mapsto$ 1, `m` $\mapsto$ 7 `ret` $\mapsto 7 + \ldots$
multiply `n` $\mapsto$ 0, `m` $\mapsto$ 7 `ret` $\mapsto$ 0

Functions: recursion

- Functions can call *themselves* in their definition.

```
# calculates n * m (in a complicated way)
def multiply(n, m):
    if n == 0:
        return 0
    else:
        return m + multiply(n - 1, m)
```

- How does the call stack look for `multiply(2, 7)` look like?

multiply n $\mapsto$ 2, m $\mapsto$ 7 ret $\mapsto 7 + \ldots$
multiply n $\mapsto$ 1, m $\mapsto$ 7 ret $\mapsto 7 + 0 = 7$

Functions: recursion

- Functions can call *themselves* in their definition.

```
# calculates n * m (in a complicated way)
def multiply(n, m):
    if n == 0:
        return 0
    else:
        return m + multiply(n - 1, m)
```

- How does the call stack look for `multiply(2, 7)` look like?

multiply
$\texttt{n} \mapsto 2$, $\texttt{m} \mapsto 7$
$\texttt{ret} \mapsto 7 + 7 = 14$

Functions: recursion

- It is crucial that the arguments of a recursive call are in some sense 'smaller' than the arguments of the function call itself.
- What happens if we write `multiply` as follows

```
def multiply(n, m):
    if n == 0:
        return 0
    else:
        return m + multiply(n, m)
```

Functions: recursion

- It is crucial that the arguments of a recursive call are in some sense ‘smaller’ than the arguments of the function call itself.
- What happens if we write `multiply` as follows

```
def multiply(n, m):
    if n == 0:
        return 0
    else:
        return m + multiply(n, m)
```

```
>>> multiply(2, 7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in multiply
  File "<stdin>", line 5, in multiply
...
  File "<stdin>", line 5, in multiply
RuntimeError: maximum recursion depth exceeded
```

- Stack overflow!

Functions: recursion

- You can also have many recursive calls

```
def fib(n):
    if n == 0:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

- Is it well defined?

Functions: recursion

- You can also have many recursive calls

```
def fib(n):
    if n == 0:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

- Is it well defined? No, what about `fib(1)` ?

Functions: recursion

- You can also have many recursive calls

```
def fib(n):
    if n == 0:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

- Is it well defined? No, what about `fib(1)` ?

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

- Is it well defined?

Functions: recursion

- You can also have many recursive calls

```
def fib(n):
    if n == 0:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

- Is it well defined? No, what about `fib(1)` ?

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

- Is it well defined? Yes.

Functions: recursion

- The argument itself *can* increase. . .

```
def reverse_string(s):
    return reverse_from_n(s, 0)

def reverse_from_n(s, i):
    if i == len(s):
        return ''
    else:
        return reverse_from_n(s, i+1) + s[i]
```

Functions: recursion

- The argument itself *can* increase...

```
def reverse_string(s):
    return reverse_from_n(s, 0)

def reverse_from_n(s, i):
    if i == len(s):
        return ''
    else:
        return reverse_from_n(s, i+1) + s[i]
```

- But if you look closer `len(s) - i` is strictly decreasing.

Functions: recursion

- Is the folowing function well defined (for $n > 0$)?

```
def collatz(n):
    if n == 1:
        return 0
    elif n % 2 == 0:
        return 1 + collatz(n/2)
    else:
        return 1 + collatz(3*n+1)
```

Functions: recursion

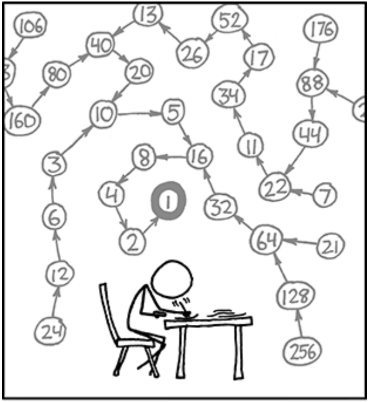
- Is the folowing function well defined (for $n > 0$)?

```
def collatz(n):
    if n == 1:
        return 0
    elif n % 2 == 0:
        return 1 + collatz(n/2)
    else:
        return 1 + collatz(3*n+1)
```

- Who knows! It has been an open problem for years.

The collatz conjecture

By the XKCD webcomic



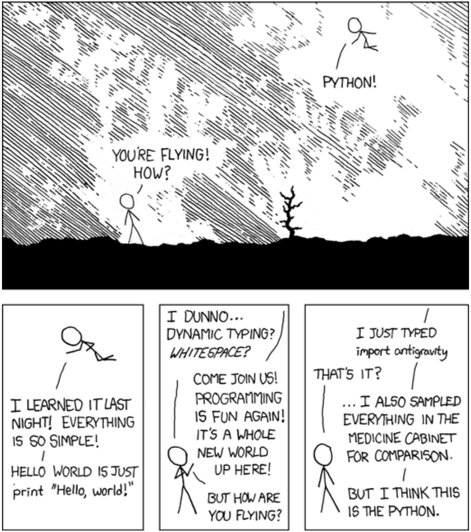
THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

Modules

- In general, programming languages come with a *library* of functions organized in some way.
- In Python, the library is organized in *modules*.
- For the moment, a module is a collection of related functions.
- Modules are used (imported) with the `import` keyword.

Python module library

By the XKCD webcomic



Using modules

- As an example we will use the `random` module. It contains functions to generate random numbers in various probability distributions.

Using modules

- As an example we will use the `random` module. It contains functions to generate random numbers in various probability distributions.
- First we need to *import* the module

```
>>> import random
```

Using modules

- As an example we will use the `random` module. It contains functions to generate random numbers in various probability distributions.
- First we need to *import* the module

```
>>> import random
```

- The functions in the module will be inside the `random` *namespace*. They are accessed using the *dot notation*

```
# Returns an integer from 1 to 10, endpoints included
>>> random.randint(1, 10)
7
```

Using modules

- As an example we will use the `random` module. It contains functions to generate random numbers in various probability distributions.
- First we need to *import* the module

```
>>> import random
```

- The functions in the module will be inside the `random` *namespace*. They are accessed using the *dot notation*

```
# Returns an integer from 1 to 10, endpoints included
>>> random.randint(1, 10)
7
```

- Suggested homework: read the book's intro to the `math` module.

Using modules

- You can import functions into the main namespace

```
>>> from random import choice
>>> choice('abcdef')
c
```

Using modules

- You can import functions into the main namespace

```
>>> from random import choice
>>> choice('abcdef')
c
```

- You can also import *everything* into the main namespace

```
>>> from random import *
```

Using modules

- You can import functions into the main namespace

```
>>> from random import choice
>>> choice('abcdef')
c
```

- You can also import *everything* into the main namespace

```
>>> from random import *
```

- But please don't! unless it is extremely necessary.

Using modules

- You can import functions into the main namespace

```
>>> from random import choice
>>> choice('abcdef')
c
```

- You can also import *everything* into the main namespace

```
>>> from random import *
```

- But please don't! unless it is extremely necessary.
- You can import modules and assign them a different name

```
>>> import random as r
>>> r.randint(1, 10)
7
```

Random numbers

By Dilbert

- Related topic in Theoretical Computer Science:
  http://en.wikipedia.org/wiki/Algorithmically_random_sequence

Repetition

- Suppose we want to make a function that given n calculates $\sum_{i=1}^{n} i$.

```
def sum_up_to(n):
    res = 1 + 2 + ... + n
    return res
```

This is not a valid program, for many reasons.

Repetition

- Suppose we want to make a function that given n calculates $\sum_{i=1}^{n} i$.

```
def sum_up_to(n):
    res = 1 + 2 + ... + n
    return res
```

  This is not a valid program, for many reasons.

- Luckily, computers are very good at doing repetitive things.
  We have the `while` statement to aid us.

```
def sum_up_to(n):
    i = 1
    v = 0
    while i <= n:
        v = v + i
        i = i + 1
    return v
```

  The body gets repeated while the condition evaluates to *true*.

Repetition

- Another handy construction is the `for` statement
- It goes through so called 'iterable' objects, e.g. strings

```
>>> for letter in 'hello':
...     print 'Give me an "' + letter + '"!'
...
Give me an "h"!
Give me an "e"!
Give me an "l"!
Give me an "l"!
Give me an "o"!
```

Repetition

- Another handy construction is the `for` statement
- It goes through so called 'iterable' objects, e.g. strings

```
>>> for letter in 'hello':
...   print 'Give me an "' + letter + '"!'
...
Give me an "h"!
Give me an "e"!
Give me an "l"!
Give me an "l"!
Give me an "o"!
```

- 'Lists' are also iterable (we will see them later)

```
>>> range(3)
[0, 1, 2]
>>> for i in range(3):
...   print i**2
...
0
1
4
```

Repetition: while loops

- `while` loops are a powerful but tricky construction.
- They can run forever and make our program hang!

```
while True:
    x = x + 1
```

Repetition: while loops

- `while` loops are a powerful but tricky construction.
- They can run forever and make our program hang!

```
while True:
    x = x + 1
```

ok, we would not write that, but what about...

```
x = int(raw_input())
sum = 0
while x != 100:
    sum = sum + x
    x = x + 2
```

Repetition: while loops

- `while` loops are a powerful but tricky construction.
- They can run forever and make our program hang!

```
while True:
    x = x + 1
```

ok, we would not write that, but what about. . .

```
x = int(raw_input())
sum = 0
while x != 100:
    sum = sum + x
    x = x + 2
```

- If $x > 100$ or x is odd this loop never ends.

Repetition: loop invariants

- A *loop invariant* is an invariant used to prove properties of loops.
- For example, correctness and termination of loops.
- Connected to pre and post-conditions.

Repetition: loop invariants

- A *loop invariant* is an invariant used to prove properties of loops.
- For example, correctness and termination of loops.
- Connected to pre and post-conditions.

E.g.: `count(c:String, sentence:String)` $\rightarrow$ `res:Int`

- `pre:` `True`
- `post:` $res = |[1 : i \in \{0, \ldots, |sentence| - 1\}, \texttt{sentence}_i = \texttt{c}]|$

Repetition: loop invariants

- A *loop invariant* is an invariant used to prove properties of loops.
- For example, correctness and termination of loops.
- Connected to pre and post-conditions.

E.g.: `count(c:String, sentence:String)` $\rightarrow$ `res:Int`

- `pre:` `True`
- `post:` $res = |[1 : i \in \{0, \ldots, |sentence| - 1\}, \mathtt{sentence}_i = \mathtt{c}]|$

Suppose we have the following implementation

```
def count(c, sentence):
    i = 0; n = 0
    while i < len(sentence):
        if sentence[i] == c: n = n + 1
        i = i + 1
    return n
```

Repetition: loop invariants

post: $res = |[1 : i \in \{0, \ldots, |sentence| - 1\}, \texttt{sentence}_i = \texttt{c}]|$

```
def count(c, sentence):
    i = 0; n = 0
    while i < len(sentence):
        if sentence[i] == c: n = n + 1
        i = i + 1
    return n
```

Repetition: loop invariants

post: $res = |[1 : i \in \{0, \ldots, |sentence| - 1\}, \texttt{sentence}_i = \texttt{c}]|$

```
def count(c, sentence):
    i = 0; n = 0
    while i < len(sentence):
        if sentence[i] == c: n = n + 1
        i = i + 1
    return n
```

Let **C** be our loop condition and **I** be our loop invariant, a theorem says:

$$\frac{\{C \wedge I\}\ \text{body}\ \{I\}}{\{I\}\ \textbf{while}\ (C)\ \text{body}\ \{\neg C \wedge I\}}$$

- **C:**

Repetition: loop invariants

post: $res = |[1 : i \in \{0, \ldots, |sentence| - 1\}, \texttt{sentence}_i = \texttt{c}]|$

```
def count(c, sentence):
    i = 0; n = 0
    while i < len(sentence):
        if sentence[i] == c: n = n + 1
        i = i + 1
    return n
```

Let **C** be our loop condition and **I** be our loop invariant, a theorem says:

$$\frac{\{C \wedge I\}\ \text{body}\ \{I\}}{\{I\}\ \textbf{while}\ (C)\ \text{body}\ \{\neg C \wedge I\}}$$

- **C:** $i < |\texttt{sentence}|$
- **I:**

Repetition: loop invariants

post: $res = |[1 : i \in \{0, \ldots, |sentence| - 1\}, \texttt{sentence}_i = \texttt{c}]|$

```
def count(c, sentence):
    i = 0; n = 0
    while i < len(sentence):
        if sentence[i] == c: n = n + 1
        i = i + 1
    return n
```

Let **C** be our loop condition and **I** be our loop invariant, a theorem says:

$$\frac{\{C \wedge I\} \text{ body } \{I\}}{\{I\} \textbf{ while } (C) \text{ body } \{\neg C \wedge I\}}$$

- **C:** $i < |\texttt{sentence}|$
- **I:** $0 \leq i \leq |\texttt{sentence}| \wedge n = |[1 : x \in \{0, \ldots, i - 1\}, \texttt{sentence}_x = \texttt{c}]|$

Repetition: loop invariants

post: $res = |[1 : i \in \{0, \ldots, |sentence| - 1\}, \texttt{sentence}_i = \texttt{c}]|$

```
def count(c, sentence):
    i = 0; n = 0
    while i < len(sentence):
        if sentence[i] == c: n = n + 1
        i = i + 1
    return n
```

Let **C** be our loop condition and **I** be our loop invariant, a theorem says:

$$\frac{\{C \wedge I\}\ \text{body}\ \{I\}}{\{I\}\ \textbf{while}\ (C)\ \text{body}\ \{\neg C \wedge I\}}$$

- **C:** $i < |\texttt{sentence}|$
- **I:** $0 \leq i \leq |\texttt{sentence}| \wedge n = |[1 : x \in \{0, \ldots, i - 1\}, \texttt{sentence}_x = \texttt{c}]|$

If we chose correctly our invariant, with $\neg C \wedge I$ we should be able to prove the postcondition.

Repetition: where's the catch?

Are the `for` and `while` statements equivalent?

Repetition: where's the catch?

Are the `for` and `while` statements equivalent?

- Short answer: in Python, yes.

Repetition: where's the catch?

Are the `for` and `while` statements equivalent?

- Short answer: in Python, yes.

Long answer:

- In old languages like BASIC and Pascal the `for` statement was meant to be used as `for i = A to B: body`. Modifications to `i` in the body would not change the iteration.
- In a `while` statement, the expression gets evaluated in every loop.

Repetition: where's the catch?

Are the `for` and `while` statements equivalent?

- Short answer: in Python, yes.

Long answer:

- In old languages like BASIC and Pascal the `for` statement was meant to be used as `for i = A to B: body`. Modifications to `i` in the body would not change the iteration.
- In a `while` statement, the expression gets evaluated in every loop.

Some facts (check this out):

- In theoretical computer science the difference between `while` and `for` statements is kept.

Repetition: where's the catch?

Are the `for` and `while` statements equivalent?

- Short answer: in Python, yes.

Long answer:

- In old languages like BASIC and Pascal the `for` statement was meant to be used as `for i = A to B: body`. Modifications to `i` in the body would not change the iteration.
- In a `while` statement, the expression gets evaluated in every loop.

Some facts (check this out):

- In theoretical computer science the difference between `while` and `for` statements is kept.
- Using what we have seen you can write *any possible program*!

Repetition: where's the catch?

Are the `for` and `while` statements equivalent?

- Short answer: in Python, yes.

Long answer:

- In old languages like BASIC and Pascal the `for` statement was meant to be used as `for i = A to B: body`. Modifications to `i` in the body would not change the iteration.
- In a `while` statement, the expression gets evaluated in every loop.

Some facts (check this out):

- In theoretical computer science the difference between `while` and `for` statements is kept.
- Using what we have seen you can write *any possible program*!
- But, if you don't use `while` you can only write 'some' of them.

Repetition: where's the catch?

Are the `for` and `while` statements equivalent?

- Short answer: in Python, yes.

Long answer:

- In old languages like BASIC and Pascal the `for` statement was meant to be used as `for i = A to B: body`. Modifications to `i` in the body would not change the iteration.
- In a `while` statement, the expression gets evaluated in every loop.

Some facts (check this out):

- In theoretical computer science the difference between `while` and `for` statements is kept.
- Using what we have seen you can write *any possible program*!
- But, if you don't use `while` you can only write 'some' of them.
- In fact, you could write any program using just ONE `while`.

References

- Chapters 3 and 5–7 of the book
  http://greenteapress.com/thinkpython/thinkpython.html
- Wikipedia article on 'Call Stack'
  http://en.wikipedia.org/wiki/Call_stack
- Wikipedia article on 'Collatz conjecture'
  http://en.wikipedia.org/wiki/Collatz_conjecture
- Wikipedia article on 'Loop invariants'
  http://en.wikipedia.org/wiki/Loop_invariant
- The `random` module
  http://docs.python.org/library/random.html