Looping and iteration

The INFDEV Team @ HR

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Lecture topics

- Repeated behaviors
- `while` statements and their semantics
- Expressive power of `while`
- Termination and infinite iteration
- Explosion of states with `while`

Repeated behaviors

Repeated behaviors

Repeated behaviors

- Sometimes running code just once is not enough
- We can *loop* execution of a block of code until some *condition* is met
- Extreme increase in expressive power

Repeated behaviors

Repeated behaviors

- While *there are hostile aliens*
- Do *attack an alien*

Repeated behaviors

Repeated behaviors

- Loops can solve very big problems
- Each step of the loop removes a part of the problem
- We typically stop when all parts of the problem have been removed

Repeated behaviors

Breaking problems up

- Problem: **kill all aliens**
- Problem piece: **a single alien to be killed**
- Solution piece: **attack a single alien**
- Termination condition: **there are no more aliens**

Repeated behaviors

Breaking problems up

- Of course loops can be combined with each other
- This means that we can *cascade* repetition
- This is the building block of *intelligent decisions* in our programs

Repeated behaviors

Breaking problems up

- While *there are hostile alien armies*
  - Do *pick an alien army*
  - While *there are aliens in the army*
    - Do *attack an alien*

Repeating behaviors in Python

Repeating behaviors in Python

`while`

- Python offers built-in facilities for repetition
- `while` statement
- We can repeat execution of a block of code

Repeating behaviors in Python

`while`

- The general form is `while CONDITION: BODY` (w_{CB})
- If the condition is true, then we jump to the beginning of `BODY`, otherwise we jump to the end of the whole `while`

$$\begin{cases} (PC, S) \overset{w_{CB}}{\rightarrow} (firstLine(B), S) & when \quad (PC, S) \overset{C}{\rightarrow} TRUE \\ (PC, S) \overset{w_{CB}}{\rightarrow} (skipAfter(B), S) & when \quad (PC, S) \overset{C}{\rightarrow} FALSE \end{cases}$$

Repeating behaviors in Python

`while`

- Remember that Python is *indentation*-based
- White-spaces go at the beginning of some lines
- A more indented line is *within* a less indented line above

Repeating behaviors in Python

`while`

- Indentation specifies where the `body` begins and ends
- The general form of a `while` is thus:
  - `while COND:`
  - `newline`
  - **indentation**
  - `code of body`
  - **de-indentation**

A correct example


```
1 n = 64
2 i = 0
3 while n > 1:
4   n = n / 2
5   i = i + 1
```

An incorrect example


```
1 n = 64
2 i = 0
3 while n > 1:
4 n = n / 2
5 i = i + 1
```

Repeating behaviors in Python

`while`

- `while` statements eventually terminate (hopefully)
- if the `condition` evaluates to `False`, then we skip after the end of the `while`

After a `while`


```
1 n = 64
2 i = 0
3 while n > 1:
4   n = n / 2
5   i = i + 1
6 print(i)
```

After a `while`?

Without indentation, this:

```
1 n = 64
2 i = 0
3 while n > 1:
4 n = n / 2
5 i = i + 1
6 print(i)
```

would be indistinguishable from both:

```
1 n = 64       1    n = 64
2 i = 0        2    i = 0
3 while n > 1:3     while n > 1:
4   n = n / 2 4       n = n / 2
5   i = i + 1 5       i = i + 1
6 print(i)     6     print(i)
```

Reasoning about `while`

Reasoning about `while`

`while`

- `while` effectively rewrites the code to become as long as the problem needs
- Until run-time, we are not really sure how long the code will become

Example `while`'s

What values of `n` will we `print`?

```
1  i = 0
2  j = 1
3  n = 0
4  while i < 10:
5    i = i + 1
6    n = n + j
7    print(n, end="␣")
```

Example `while`'s

What values of `n` will we `print`?

```
1  i = 0
2  j = 1
3  n = 0
4  while i < 10:
5    i = i + 1
6    n = n + j
7    print(n, end="␣")
```

1 2 3 4 5 6 7 8 9 10

Example `while`'s

What values of `n` will we `print`?

```
1  i = 0
2  j = 2
3  n = 0
4  while i < 10:
5    i = i + 1
6    n = n + j
7    print(n, end="␣")
```

Example `while`'s

What values of `n` will we `print`?

```
1  i = 0
2  j = 2
3  n = 0
4  while i < 10:
5    i = i + 1
6    n = n + j
7    print(n, end="␣")
```

2 4 6 8 10 12 14 16 18 20

Reasoning about `while`

`while`

- Each iteration produces new values of the variables
- These new values are then fed back into the next iteration
- Eventually these cause the condition to become false
  - Or the program runs forever and never produces a result

Reasoning about `while`

`while`

- Each iteration produces new values of the variables
- These new values are then fed back into the next iteration
- Eventually these cause the condition to become false
  - Or the program runs forever and never produces a result
  - We'd rather not have this one

Example `while`'s

i	j	n
0	2	0

```
1  while i < 10:
2    i = i + 1
3    n = n + j
4    print(n, end="␣")
```

Example `while`'s

i	j	n
0	2	0

```
1  while i < 10:
2    i = i + 1
3    n = n + j
4    print(n, end="␣")
```

i	j	n
1	2	2

Example `while`'s

i	j	n
1	2	2

```
1  while i < 10:
2    i = i + 1
3    n = n + j
4    print(n, end="␣")
```

Example while's

i	j	n
1	2	2

```
1  while i < 10:
2    i = i + 1
3    n = n + j
4    print(n, end="␣")
```

i	j	n
2	2	4

Example `while`'s

i	j	n
2	2	4

```
1  while i < 10:
2    i = i + 1
3    n = n + j
4    print(n, end="␣")
```

Example while's

i	j	n
2	2	4

```
1  while i < 10:
2    i = i + 1
3    n = n + j
4    print(n, end="␣")
```

i	j	n
3	2	6

Example `while`'s

After `k` iterations (for `k < 10`):

`i`	`j`	`n`
`k`	`2`	`2 * k`

```
1 while i < 10:
2   i = i + 1
3   n = n + j
4   print(n, end="␣")
```

Example `while`'s

After `k` iterations (for `k < 10`):

i	j	n
k	2	2 * k

```
1 while i < 10:
2   i = i + 1
3   n = n + j
4   print(n, end="␣")
```

i	j	n
k+1	2	2 * k + 2

Example `while`'s

After `k` iterations (for `k = 10`):

i	j	n
10	2	20

```
1 while i < 10:
2   i = i + 1
3   n = n + j
4   print(n, end="␣")
```

Example `while`'s

After `k` iterations (for `k = 10`):

i	j	n
10	2	20

```
1 while i < 10:
2   i = i + 1
3   n = n + j
4   print(n, end="␣")
```

We jump to the first instruction **after** the while loop, and do not touch the state further.

i	j	n
10	2	20

Reasoning about `while`

Readability and termination

- The loop above is *well designed*
- All iterations produce a new piece of a logical series
  - (The `j`-th row of the table of multiplication)

Reasoning about `while`

Readability and termination

- After each iteration we know that we have `i` elements correctly computed
- After each iteration we know that we have `10-i` elements still to compute
- When `i=10` then we have 10-10 elements still to compute
  - This is the **termination condition**
  - Since `i` keeps growing, we know that eventually the termination condition will be met

Reasoning about `while`

Nesting

- Loops can be nested
- A loop can be inside a loop (which can further be inside other loops)
- This makes it slightly harder to reason

Example nested `while`'s


```
1  j = 1
2  while j <= 10:
3    i = 0
4    n = 0
5    while i < 10:
6      i = i + 1
7      n = n + j
8      print(n, end="\t")
9    j = j + 1
10   print("")
```

Example nested `while`'s

We now know that the semantics of the inner loop is **print the j-th row of the table of multiplication**, so instead of reasoning on:

```
1  j = 1
2  while j <= 10:
3    i = 0
4    n = 0
5    while i < 10:
6      i = i + 1
7      n = n + j
8    print(n, end="\t")
9    j = j + 1
10   print("")
```

we reason on:

```
1  j = 1
2  while j <= 10:
3    print the j-th row of the table of multiplication
4    j = j + 1
5    print("")
```

Example nested `while`'s

j	output
1	nothing

```
1  while j <= 10:
2    print the j-th row of the table of multiplication
3    j = j + 1
4    print("")
```

Example nested `while`'s

j	output
1	nothing

```
1  while j <= 10:
2    print the j-th row of the table of multiplication
3    j = j + 1
4    print("")
```

j	output
2	1st row of table

Example nested `while`'s

j	output
2	1st row of table

```
1  while j <= 10:
2    print the j-th row of the table of multiplication
3    j = j + 1
4    print("")
```

Example nested `while`'s

j	output
2	1st row of table

```
1 while j <= 10:
2   print the j-th row of the table of multiplication
3   j = j + 1
4   print("")
```

j	output
3	1st and 2nd rows of table

Example nested `while`'s

...

Example nested `while`'s

Thus for all `k`'s such that `k` ≤ 10:

`j`	`output`
`k`	`first k-1 rows of table`

```
1 while j <= 10:
2   print the j-th row of the table of multiplication
3   j = j + 1
4   print("")
```

Example nested `while`'s

Thus for all `k`'s such that `k` ≤ 10:

j	output
k	first k-1 rows of table

```
1 while j <= 10:
2   print the j-th row of the table of multiplication
3   j = j + 1
4   print("")
```

j	output
k+1	first k rows of table

Example nested `while`'s

Eventually we get `j = 11`:

j	output	PC
11	first 10 rows of table	1

```
while j <= 10:
  print the j-th row of the table of multiplication
  j = j + 1
  print("")
...
```

1
2
3
4
5

Example nested `while`'s

Eventually we get `j = 11`:

j	output	PC
11	first 10 rows of table	1

```
while j <= 10:
  print the j-th row of the table of multiplication
  j = j + 1
  print("")
...
```

1
2
3
4
5

j	output	PC
11	first 10 rows of table	5

Termination (or lack thereof)

Termination (or lack thereof)

Wait! It gets worse!

- It is not guaranteed that a loop will terminate
- A loop that does not terminate gets the program stuck forever
- It is a bit sad for the machine
- Care when designing loops is needed to prevent this

Termination (or lack thereof)

Care in the design

- A loop changes the state of the program many times
- A **good** loop changes the state in one **direction**
- Every step should bring us closer to the **final state**
- The condition defines the aspects of the final state

Example non terminating `while`


```
1  i = 1
2  while i > 0:
3    i = i + 1
4    print(i)
```

What is the issue here?

Example non terminating `while`


```
1  i = 1
2  while i > 0:
3    i = i + 1
4    print(i)
```

What is the issue here?

Iterations do not go **towards** the condition, but **away** from it.

Example non terminating `while`

This is not a duplicated slide.

```
1  i = 1
2  while i < 10:
3    i = i + 1
4    print(i)
```

What is the issue here?

Example non terminating `while`

This is not a duplicated slide.

```
1  i = 1
2  while i < 10:
3    i = i + 1
4    print(i)
```

What is the issue here?

Iterations are **orthogonal (unrelated)** to the condition.

No iteration changes elements tested in the condition.

Exponential explosion of potential control-paths

Exponential explosion of potential control-paths

Exponential explosion of potential control-paths

- How many things can happen in a `while` loop?
- Depends on its content

Exponential explosion of potential control-paths

Exponential explosion of potential control-paths

- The more we nest loops and conditionals within a loop...

Exponential explosion of potential control-paths

Exponential explosion of potential control-paths

- The more we nest loops and conditionals within a loop...
- ...the more things may happen at run-time

Loops with nested `if`'s


```
1 while C0:
2   if C1:
3     A1
4   else:
5     B1
```

How many execution paths per **N** iterations of the loop?

Loops with nested `if`'s


```
1 while C0:
2   if C1:
3     A1
4   else:
5     B1
```

How many execution paths per **N** iterations of the loop?

2 execution paths per iteration

2^N execution paths per `N` iterations

Loops with nested `if`'s


```
1 while C0:
2   if C1:
3     A1
4   else:
5     B1
```

Example: execution paths per **3** iterations of the loop.

```
A1 A1 A1
A1 A1 B1
A1 B1 A1
A1 B1 B1
B1 A1 A1
B1 A1 B1
B1 B1 A1
B1 B1 B1
```

Exponential explosion of potential control-paths

Exponential explosion of potential control-paths

- Consider a loop that performs `m` iterations
- With `n` `if`'s inside
- Each iteration can have 2^n possible execution paths
- The whole loop can have $2^{n \times m}$ possible execution paths[a]

[a] $2^{2 \times 10} = 1048576$

Exponential explosion of potential control-paths

Exponential explosion of potential control-paths

- Each path can alter the state in a different way
- After a `while` with many billions possible paths
  - We have many billions possible resulting states

Exponential explosion of potential control-paths

Exponential explosion of potential control-paths

- The more nested the code inside a `while`
- The more complex its behavior
- *The harder it is to reason about it!*

Exponential explosion of potential control-paths

The value of reasoning

- **Always keep in mind:**
- You have the power to make your own life a living Hell...

Exponential explosion of potential control-paths

The value of reasoning

- **Always keep in mind:**
- You have the power to make your own life a living Hell...
- ...unless you reason first and then structure code logically

Assignment

Assignment

Drawing with loops!

- Exclusively with `while`-loops
- Reading the appropriate parameters from the command line (figure height, width, etc.)
- Draw the following figures on the console:
  - A full square
  - The border of a square
  - A rectangle triangle
  - A centered, isosceles triangle
  - A circle
  - A smiley face
- **See assignment description online for examples and better details**

This is it!

The best of luck, and thanks for the attention!