ECOR 1041
Computation and Programming

Control Flow: Repetition (Loop) Statements

Carleton University

References

- *Practical Programming*, 3rd ed., Chapter 9
  - *Looping Until a Condition is Reached*, pp. 160 - 162
  - *Looping Over a Range of Numbers*, to the end of section *Generating Ranges of Numbers*, pp. 152 - 154

Lecture Objectives

- Introduce and apply these repetition (loop) statements: `while` and `for`

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - Control flow, loop (repetition)
  - `while` and `for` keywords
  - Built-in `range` function

Learning Outcomes

- Be able to design and implement functions that use `while` and `for` loops to repeatedly execute blocks of code

Recap: Control Flow (Flow of Control)

- *Control flow*: the order in which statements in a program (written in an imperative programming language, like Python) are executed
- *Control flow statements* choose between two or more paths of execution through the code
- In this course, the control flow statements we will use are *conditional* statements and *loop (repetition)* statements
- In this lecture, we will introduce Python's `while` and `for` loops

Example: Calculating Factorials

- $n!$ ("factorial n") is defined as:

  $$n! = n(n - 1) \cdots 2 \cdot 1, n > 0$$

- To calculate $n!$, we need to perform n-1 multiplications:

  $$1 \times 2 \times 3 \times . . . \times n$$

Example: Calculating Factorials

- We could calculate *n*! this way:

```
fact_n = 1
i = 2
fact_n = fact_n * i    # 1 * 2
i += 1
fact_n = fact_n * i    # 1 * 2 * 3
i += 1
...
i += 1  # i now equals n
fact_n = fact_n * i    # 1 * 2 * 3 * ... * n
```

Drawbacks to this Technique

- This code always calculates *n*! for one value of *n*
  - `fact_n = fact_n * i` is executed *n*-1 times
- We have to add/delete statements to calculate the factorial of different values of *n*

Control Flow: Repetition

- We need a way to repeatedly perform

```
fact_n = fact_n * i
```

  for the sequence `i` = 2, 3, 4, … `n`
- Generalizing this: we need a control flow construct that causes Python to repeatedly execute a block of code
- Python provides two statements that support repetition: the **`while` loop** and the **`for` loop**

`while` Statement

- Syntax:

```
while condition:
    block
```

- Semantics: if *`condition`* is `True`, execute *`block`* once, then reevaluate *`condition`*. If *`condition`* is `True`, execute *`block`* again, then reevaluate *`condition`*. If *`condition`* is `False`, exit the loop; that is, execute the statement after the `while` statement

`factorial` Function

```
def factorial(n: int) -> int:
    """Return n!

    Precondition: n > 0.
    """
    fact_n = 1
    i = 2
    while i <= n:
        fact_n = fact_n * i
        i += 1
    return fact_n
```

`factorial` Function

- Condition `i <= n` is `True` when `i` equals 2, 3, 4, ..., `n`, so the loop body

```
fact_n = fact_n * i
i += 1
```

  is executed `n-1` times
- During the last iteration, the value of `i` is becomes equal to `n+1`, the condition is `False` the next time it is evaluated, and the loop is exited

`range` Function

- Python's built-in `range` function returns an object that generates a sequence of integers
- `range(n)` returns an object that generates the sequence: `0`, `1`, `2`, ..., `n-1`
  - Note that the last value is `n-1`, not `n`
- `range(a, b)` (`a` and `b` are integers) returns an object that generates the sequence: `a`, `a+1`, `a+2`, ..., `b-1`

`for` Statement

- The `for` statement iterates over the elements of a sequence
- Syntax:

```
for variable in sequence:
    block
```

`for` Statement

- Semantics:
  - *`variable`* is assigned the first element in *`sequence`*, then *`block`* is executed
  - *`variable`* is assigned the second element in *`sequence`*, then *`block`* is executed, and so on, until…
  - *`variable`* is assigned the last element in *`sequence`*, then *`block`* is executed

`factorial` Function, Version 2

- We can rewrite the factorial function, replacing the `while` loop with a `for` loop

```
def factorial(n: int) -> int:
    """Return n!

    Precondition: n > 0.
    """
    fact_n = 1
    for i in range(2, n + 1):
        fact_n = fact_n * i
    return fact_n
```

`factorial` Function, Version 2

- `range(2, n + 1)` returns an object that generates the sequence 2, 3, 4, ..., `n`
- `for i in range(2, n + 1)` assigns `i` the values 2, 3, 4, ..., `n`
- Every time `i` is assigned the next value in the sequence the loop body is executed, so `fact_n = fact_n * i` is executed a total of `n-1` times

Looping Until a Condition is Reached

- `for` loops cannot be used when we do not know in advance how many iterations are required; that is, how many times the loop body should be executed
- For these cases, use a `while` loop

Calculating Square Roots

- Write a function that returns an approximation of the positive square root of a non-negative number, x
- An approximate solution g is "close enough" to the square root of x if $g * g \approx x$; that is, abs($g * g - x$) < ϵ, where ϵ is a small constant value

Heron's Algorithm

- To calculate the square root of a non-negative number, *x*:
  - Start with a guess, *g*
  - If *g* * *g* is not "close enough" to *x*, create a new guess by averaging *g* and *x*/*g*, assign the new guess to *g*
  - Repeat the previous step until *g* * *g* is "close enough" to *x*

Heron's Algorithm: Example

- Calculate the positive square root of 25
- Use 0.001 as ϵ
- Start by assigning g the guess 25 / 4 ⇒ 6.25

Heron's Algorithm: Example

- Iteration 1
  - abs(6.25 * 6.25 - 25) ⇒ abs(39.0625 - 25) > 0.001
    - 6.25 is not "close enough" to the square root of 25
    - Assign *g* the new guess (6.25 + 25 / 6.25) / 2 ⇒ 5.125

Heron's Algorithm: Example

- Iteration 2
  - abs(5.125 * 5.125 - 25) ⇒ abs(26.265625 - 25) > 0.001
    - 5.125 is not "close enough" to the square root of 25, but it is closer than the previous guess
    - Assign *g* the new guess (5.125 + 25 / 5.125) / 2 = 5.001
- Iteration 3
  - abs(5.001*5.001 - 25) = 0.01; g = (5.001 + 25/5.001) / 2 = 5.000 …

Heron's Algorithm: Example

- By repeating these steps, eventually *g* is assigned a value such that abs(*g* * *g* - 25) < 0.001; that is, *g* will be approximately 5, which will be considered to be close enough to the square root of 25

heron Function

```
EPSILON = 0.001
def heron(x: float) -> float:
    """Return the positive square root of x.

    Precondition: x > 0.
    """
    guess = x / 4
    while abs(guess * guess - x) >= EPSILON:
        guess = (guess + x / guess) / 2
    return guess
```

Recap of Learning Outcomes

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - Control flow, loop (repetition)
  - `while` and `for` keywords
  - Built-in `range` function

Learning Outcomes

- Be able to design and implement functions that use `while` and `for` loops to repeatedly execute blocks of code