Computational Structures in Data Science

Lecture 4: Sequences and `for` Loops

Week 2, Summer 2024. 6/24 (Mon)

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

Welcome to Week 2!

Upcoming due dates (11:59 pm PST)

- Homework 01: Due July 24th
- Lab 01: Due July 24th

Announcements

Exam dates

Midterm: Wednesday July 17th, 3PM – 5PM PST
Final: Wednesday August 7th, 3PM – 5PM PST

Exams will be **administered online**, and **proctored via Zoom**. You may need to present your ID (eg student CalID card, or any ID with your name + photo) during the Zoom call to proctors.

Important: for those that can't make the above exam times, we will have **alternate exam times**. Stay tuned for details here!

Announcements

(Reminder) tips for success in Data C88C

- Attend Lab, Office Hours
- Keep on top of due dates by checking Gradescope regularly
  - This summer course moves **QUICKLY**, don't get left behind
  - Tip: google calendar / task spreadsheet for organizing
- Ask (lots of) questions on Ed
  - **Tip**: if you'd like additional practice problems (and ask nicely on Ed!), course staff may be willing to help out!

Overview

Today:

- While loops
- For loops
- Sequences
  - Lists
- (preview) List comprehensions

Computational Structures in Data Science

Iteration with `while` Loops

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Use a while loop to repeat some task.
- Write an expression to control when a while loop stops executing

while Statement – Iteration Control

- Repeat a block of statements until a *predicate expression* is not satisfied
- At the "end" of the body, we re-evaluate the expression, and continue as long as it True
- Like conditionals and functions, we indent the body one level

```
<initialization statements>
while <predicate expression>:
    <body statements>

<rest of the program>
```


Sum The Numbers

- This is a task we'll see many times!
- The sum of 1 to 10 (inclusive) is 55. A useless, but useful, fact.

```
total = 0
n = 1
while n <= 10:
    total += n
    n += 1
print(total)
```


While Loops and Text

- Index is the name used to track a position in some sequence.
- We can "index into" a string to get an individual letter
- text[0] == "H"

```
text = "Hello, C88C!"
index = 0
while index < len(text):
        print(text[index])
        index += 1 # Same as index = index += 1
```


Sum The Numbers As a Function

```
def sum_to_n(n):
    """
    >>> sum_to_n(10)
    55
    """
    total = 0
    i = 1
    while i <= n:
        total += i
        i += 1
    return total
```


Sum The Numbers As a Function

```
def sum_to_n_down(n):
    """
    >>> sum_to_n_down(10)
    55
    """
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total
```


Computational Structures in Data Science

`for` Loops

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives: Using Lists in Practice

- `for` Loops are a ”generic” way to iterate over data.
- Compare a `for` loop and a `while` loop.
- Learn to use `range()`
- Use a string as a sequence of letters

REVIEW: while statement – iteration control

- Repeat a block of statements until a predicate expression is satisfied

```
<initialization statements>
while <predicate expression>:
    <body statements>

<rest of the program>
```

```
# Equivalent to a for loop:
text = "Hello, C88C!"
index = 0
while index < len(text):
    letter = text[index]
    print(letter)
    index += 1
```


`for` Statement – Iteration Control

- Repeat a block of statements for a structured sequence of variable bindings

```
<initialization statements>
for <variables> in <sequence expression>:
  <body statements>

<rest of the program>
```


Live Coding Demo

```
text = "Hello, C88C!"
index = 0
while index < len(text):
    letter = text[index]
    print(letter)
    index += 1

for letter in text:
    print(letter)
```

Live Coding Demo

```
index = 0
while index < 10:
    print(index)
    index += 1

for index in range(0, 10):
    print(index)
```


Summing 1 to N (Again)

```
def sum_to_n(n):
        total = 0
        for num in range(0, n + 1):
                total += num
        return total

def sum_to_n_down(n):
        total = 0
        for num in range(n, 0, -1):
                total += num
        return total
```

```
def sum_to_n_while(n):
    total = 0
    i = 1
    while i <= n:
        total += i
        i += 1
    return total

def sum_to_n_down_while(n):
    total = 0
    i = n
    while i > 0:
        total += i
        i -= 1
    return total
```


Summing 1 to N (Again)

```
def sum_to_n(n):
        total = 0
        for num in range(0, n + 1):
                total += num
        return total

def sum_to_n_down(n):
        total = 0
        for num in range(n, 0, -1):
                total += num
        return total
```

```
def sum_to_n_while(n):
    total = 0
    i = 1
    while i <= n:
        total += i
        i += 1
    return total

def sum_to_n_down_while(n):
    total = 0
    i = n
    while i > 0:
        total += i
        i -= 1
    return total
```

Tip: you can often write a **for** loop as a **while** loop

Computational Structures in Data Science

Sequences

Berkeley
UNIVERSITY OF CALIFORNIA

Sequences [Docs]

- The term sequence refers generally to a data structure consisting of an indexed collection of values, which we'll generally call elements.
  - That is, there is a first, second, third value (which CS types call #0, #1, #2, etc.). "Zero-based" vs "One-based" indexing
- A sequence may be finite (with a length) or infinite.
- It may be mutable (elements can change) or immutable.
- It may be indexable: its elements may be accessed via selection by their indices.
- It may be iterable: its values may be accessed sequentially from first to last.

<sequence expression> — What's that?

- Common sequences:
  - range() – give me all the numbers
  - Strings, e.g, "Hello, C88C!"
    - What is it a sequence of? Characters!
  - lists (next!)
- We'll start with two basic facts:
  - `range(10)` is the numbers 0 to 9, or range(0, 10)
  - `[]` means "indexing" an item in a sequence.
  - `"Hello"[0] == "H"`

Common Sequences

- *There are many types of sequences* in Python.
  - `range`
  - `string` (text data)
  - `list`
  - `tuple`
- Sequences all share some common properties.

`range`

- `range()` is a built in Python tool that generates a sequence of numbers.
  - It does not return a list unless we explicitly ask for one.
- It has many options: start, stop, and step.
- (Fun fact) Range is *lazy!* It can be iterated over, but doesn't compute all its values at once.
- **GOTCHA:** Range is exclusive in the last value!
  - **`range(10)` is a sequence on 10 numbers from 0 to 9.**

Sequence Operations

Operation	Result
`x in s`	True if an item of *s* is equal to *x*, else False
`x not in s`	False if an item of *s* is equal to *x*, else True
`s + t`	the concatenation of *s* and *t*
`s * n or n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Live Coding Demo

```
sum(range(0, 11))

def sum_to_n(n):
        return sum(range(0, n + 1))


text = 'Hello, C88C!'
len(text)
text.count('l')
text.count(8)
text.count('8')
```


Computational Structures in Data Science

Lists

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Lists are a new data type in Python.
- Lists can store any kind of data and be any length.
- We start counting items of lists at 0.
- Lists are *mutable.* We can change their data!

Lists

- A structure in Python that can hold many elements
  - Also referred to an an “array” in other programming languages.
- Lists are used to group similar items together.
  - A “contact list”, a “list of courses”, a “to do list”
- Python lists are *really* flexible!
  - Can contain any type of data
  - Can mix and match types!
  - Can add and delete items

Types We've Learned So Far

- Each *type* of data has a specific set of functions (methods) you can apply to them, and certain properties you can access.
- `int` / Integers
  - `1, -1, 0, …`
- `float` ("decimal numbers")
  - `1.0, 3.14159, 20.0`
- `string`
  - `"Hello, CS88"`
- `function`
  - `max(), min(), print(), your own functions!`
- **`list`**
  - **`['CS88', 'DATA8', 'POLSCI2', 'PHILR1B']`**

List Operations [Python Docs!]

- `[]` "square brackets": Used to access items in a list. We start at 0!
- len(): The number of items in a list
- +: We can add lists together
- min(), max(): Functions that take in a list and return some info.
- Converting between types: Strings and Lists:
  - `<string>.split(<separator>)` → List of strings
    - `'I am taking CS88.'.split(' ')`
  - `<string>.join(<list>)` → String, with the items of a list joined together.
    - `' '.join(['I', 'am', 'taking', 'C88C.'])`
- Lots more interesting tools!

Selecting Elements From a List (A Reference, Don't Memorize Yet!)

- **Selection** refers to extracting elements by their index.
- **Slicing** refers to extracting subsequences.
- These work uniformly across sequence types.

```
L = [2,0,9,10,11]
S  =  "Hello, world!"
L[2]== 9
L[-1] == L[len(t)-1] == 11
S[1] == "e" # Each element of a string is a one-element string.
L[1:4] == (L[1], L[2], L[3]) == (0, 9, 10)
S[1:2] == S[1] == "e"
S[0:5] == "Hello", S[0:5:2] == "Hlo", S[4::-1] == "olleH"
```


Rules of Indexing & Slicing

- We start counting from 0.
  - You *will* mess this up. We all do. It's ok.
  - There's lots of bad dad jokes about this. ☺
- Python provides flexibility but can be confusing.
  - `[0]` means the first item
  - `[-1]` means the last item, [-2] 2nd to last, and so on
- **Slicing: The last value is *exclusive!***
  - `[:stop], e.g. my_list[:5] # items 0-4`
  - `[start:stop], e.g. my_list[2:5] # items 2,3,4`
  - `[start:stop:step] e.g. my_list[0:8:2] # items 0,2,4,6`

Sequence Operations (Review and Reference)

Operation	Result
`x in s`	True if an item of *s* is equal to *x*, else False
`x not in s`	False if an item of *s* is equal to *x*, else True
`s + t`	the concatenation of *s* and *t*
`s * n or n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Computational Structures in Data Science

Demo: putting it all together

Berkeley
UNIVERSITY OF CALIFORNIA

Demo: putting it all together

```
def is_even(num):
    """Returns True if the input integer is even, False otherwise."""
    return num % 2 == 0
def get_every_other_letter_of_each_word(input_string):
    """For each word (separated by spaces), returns every other letter.
    Example:
    >>> get_every_other_letter_of_each_word("Hello I am Eric")
    [['H', 'l', 'o'], ['I'], ['a'], ['E', 'i']]
    """
    words = input_string.split(" ")
    output = []
    for word in words:
        # grab every other letter in word
        letters = []
        for ind in range(len(word)):
            if is_even(ind):
                # only keep letters with even indices
                letters.append(word[ind])
        output.append(letters)
    return output
```

Demo: putting it all together

```
def is_even(num):
    """Returns True if the input integer is even, False otherwise."""
    return num % 2 == 0
def get_every_other_letter_of_each_word(input_string):
    """For each word (separated by spaces), returns every other letter.
    Example:
    >>> get_every_other_letter_of_each_word("Hello I am Eric")
    [['H', 'l', 'o'], ['I'], ['a'], ['E', 'i']]
    """
    words = input_string.split(" ")
    output = []
    for word in words:
        # grab every other letter in word
        letters = []
        for ind in range(len(word)):
            if is_even(ind):
                # only keep letters with even indices
                letters.append(word[ind])
        output.append(letters)
    return output
```

Tip: there are lots of ways to implement this function, using techniques we've learned so far. Examples:

- Use a while loop instead of a for loop?
- Use string slicing ([i:j:k]) instead of the inner for loop?
- Refactor to use another helper function(s)?
- ...

Computational Structures in Data Science

(preview) List Comprehensions

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- List comprehensions let us build lists "inline".
- List comprehensions are an *expression that returns a list.*
- We can easily “filter” the list using a conditional expression, i.e. `if`

Data-driven iteration

- describe an expression to perform on each item in a sequence
- let the data dictate the control
- In some ways, nothing more than a concise for loop.

```
[ <expr with loop var> for <loop var> in <sequence expr > ]
```

```
[ <expr with loop var> for <loop var> in <sequence expr >
if <conditional expression with loop var> ]
```


Example: List comprehension

```
my_nums = [1, 2, 3, 4, 5]
print(f"my_nums: {my_nums}")
# list comprehension
my_squared_nums = [num ** 2 for num in my_nums]
print(f"my_squared_nums: {my_squared_nums}")
# list comprehension with filter
my_squared_even_nums = [num ** 2 for num in my_nums if (num % 2) == 0]
print(f"my_squared_even_nums: {my_squared_even_nums}")

Outputs:

my_nums: [1, 2, 3, 4, 5]
my_squared_nums: [1, 4, 9, 16, 25]
my_squared_even_nums: [4, 16]
```


Overview. Any Questions?

Today:

- While loops
- For loops
- Sequences
  - Lists
- (preview) List comprehensions