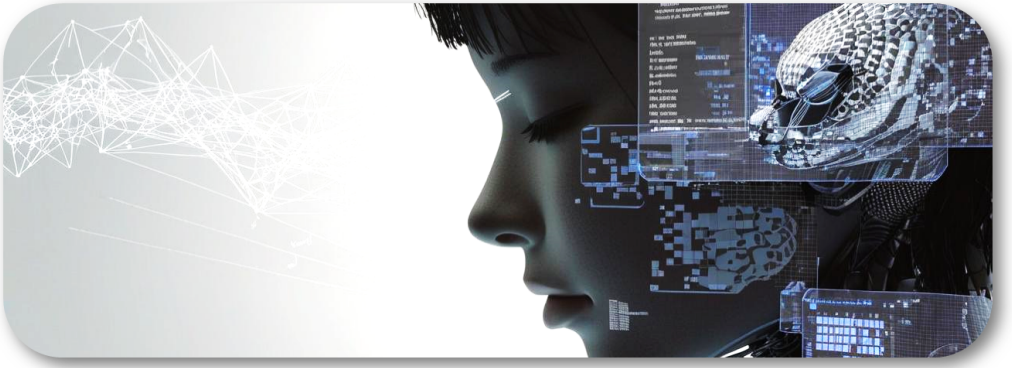


COMP 1023 Introduction to Python Programming

Tutorial 6: Collections - Lists & Sequences

COMP 1023 Teaching Team

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Quick Refresher

- Collections are fundamental data structures in Python that allow you to store and manage multiple items efficiently.
- Types of common collections:
  - `list`: Ordered & mutable
  - `tuple`: Ordered & immutable (to be taught)
  - `set`: Unordered & Mutable (self-study)
  - `dict`: Ordered & mutable **key-value pairs** (to be taught)

Constructing Lists

- Refer to lecture slides
  - `list` basics
- Construct a `list` element-by-element: `["good", "guy"]`
- Convert an iterable into a `list`: `list(<iterable>)`

Exercise: Constructing Lists

Describe the following expressions.

1. `[1.4, 2.3, 5.5]`
2. `[]`
3. `list()`
4. `list(list())`
5. `["1023",]`
6. `list(range(10))`
7. `list("comp1023")`
8. `[[0, 1], "01", 1.023]`

Exercise: Constructing Lists

Describe the following expressions.

1. `[1.4, 2.3, 5.5]`: a `list` of 3 `floats`
2. `[]`: an empty `list`
3. `list()`: an empty `list`
4. `list(list())`: an empty `list`
5. `["1023",]`: a list containing one `str`
6. `list(range(10))`: a `list` containing the `ints` from 0 (inclusive) to 10 (exclusive)
7. `list("comp1023")`: `['c', 'o', 'm', 'p', '1', '0', '2', '3']`
   - `str`s are iterated character-by-character (we will talk about iteration later)
8. `[[0, 1], "01", 1.023]`: a `list` containing a `list` of 2 `ints`, then a `str`, then a `float`
   - List elements can be of mixed types, including `list` objects

iPRS Question 1

The lists `my_list1` and `my_list2` contain the same values.

```
my_list1 = [1, 3, 5]
my_list2 = []
for i in range(1, 6, 2):
    my_list2.append(i)
```

A. True

B. False

iPRS Question 1 - Answer

The lists `my_list1` and `my_list2` contain the same values.

```
my_list1 = [1, 3, 5]
my_list2 = []
for i in range(1, 6, 2):
    my_list2.append(i)
```

A. True

B. False

Explanation

The `range(1, 6, 2)` object creates a sequence of 3 numbers: 1, 3 and 5.
Appending them to the original empty list `my_list2` creates a `list` of `[1, 3, 5]`, which contains the same values as `my_list1`.

Indexing Sequences

- Refer to lecture slides
  - `list` demonstration
  - Index operator `[]`
  - Negative numbers as indexes
- Syntax: `my_list[index]`, where `index` is an `int`
- When used as part of an assignment statement, the left-hand-side sequence needs to be mutable
- If the index refers to an nonexistent element, an `IndexError` occurs
- Types that can be indexed
  - `list` (mutable)
  - `str` (deep immutable)
  - `range` (deep immutable)
  - ...

Exercise: Indexing Sequences

Evaluate (or describe if no output) the following indexing expressions.

1. `["Python", "is", "fun"][]`
2. `["but", "real", "Python", "is", "not", "fun"][2]`
3. `["but", "real", "Python", "is", "not", "fun"][-1]`
4. `["but", "real", "Python", "is", "not", "fun"][1023]`
5. `["but", "real", "Python", "is", "not", "fun"][-7]`
6. `[["a", "b", "c"], "d"][0]`
7. `[["a", "b", "c"], "d"][0][-1]`
8. `"COMP 1023"[-4]`
9. `range(10)[-4]`
10. `["COMP", "1023"][-1][0]`
11. `a = [0]; a[0] = 1`
12. `a = [0]; a[1] = 1`
13. `a = range(1); a[0] = 10`
14. `a = "0"; a[0] = 1`

Exercise: Indexing Sequences

Evaluate (or describe if no output) the following indexing expressions.

1. `["Python", "is", "fun"][]`: SyntaxError
2. `["but", "real", "Python", "is", "not", "fun"][2]`: `"Python"`
3. `["but", "real", "Python", "is", "not", "fun"][-1]`: `"fun"`
   - Retrieves the last element
4. `["but", "real", "Python", "is", "not", "fun"][1023]`: IndexError
   - The index is way beyond the sequence end
5. `["but", "real", "Python", "is", "not", "fun"][-7]`: IndexError
   - The negative index is beyond the sequence start
6. `[["a", "b", "c"], "d"][0]`: `["a", "b", "c"]`
   - Indexing only acts on the outermost sequence
7. `[["a", "b", "c"], "d"][0][-1]`: `"c"`
   - You can chain indexing

Exercise: Indexing Sequences

Evaluate (or describe if no output) the following indexing expressions.

8. `"COMP 1023"[-4]`: `"1"`
   - You can index `str`s; the result is a `str` containing the specified character
   - Note: There is no dedicated 'character' type, unlike some other programming languages
9. `range(10)[-4]`: `6`
   - You can index `range`s
10. `["COMP", "1023"][-1][0]`: `"1"`
11. `a = [0]; a[0] = 1`: `a = [1]`
12. `a = [0]; a[1] = 1`: `IndexError`
    - You cannot add items to a `list` using out-of-range indices
13. `a = range(1); a[0] = 10`: `TypeError: 'range' object does not support item assignment`
    - A `range` is deep immutable
14. `a = "0"; a[0] = 1`: `TypeError: 'str' object does not support item assignment`
    - A `str` is deep immutable

Slicing Sequences

- Refer to lecture slides
  - `list` slicing
  - Default values
  - Edge cases
- Refer to appendix for optional details
  - There is also a nice visualization that you may learn to use
- Syntax: `sequence[start:stop:step]`, where `start`, `stop`, and `step` are `int`s
- When used as part of an assignment statement, the left-hand-side sequence needs to be mutable
- Unlike indexing, out-of-range indices are handled gracefully, and never trigger an IndexError
- Types that can be sliced: if a type can be indexed, it almost always can also be sliced
- Returns the same type as the original sequence

Exercise: Slicing Sequences for Retrieval

Evaluate the following slicing expressions.

1. `["Python", "is", "very", "fun!"][1:2]`
2. `["Python", "is", "very", "fun!"][1:]`
3. `["Python", "is", "very", "fun!"][::]`
4. `["Python", "is", "very", "fun!"][::2]`
5. `["Python", "is", "very", "fun!"][::-2]`
6. `["Python", "is", "very", "fun!"][1023:1023]`
7. `["Python", "is", "very", "fun!"][-1023:1023]`
8. `["Python", "is", "very", "fun!"][1023:-1023]`
9. `["Python", "is", "very", "fun!"][1023:-1023:-1]`
10. `["Python", "is", "very", "fun!"][::0]`
11. `"Python is very fun!"[::-1]`
12. `range(1023)[:1021]`

Exercise: Slicing Sequences for Retrieval

Evaluate the following slicing expressions.

1. `["Python", "is", "very", "fun!"][1:2]`: `["is"]`
2. `["Python", "is", "very", "fun!"][1:]`: `["is", "very", "fun!"]`
3. `["Python", "is", "very", "fun!"][::]`: `["Python", "is", "very", "fun!"]`
4. `["Python", "is", "very", "fun!"][::2]`: `["Python", "very"]`
5. `["Python", "is", "very", "fun!"][::-2]`: `["fun!", "is"]`
6. `["Python", "is", "very", "fun!"][1023:1023]`: `[]`
7. `["Python", "is", "very", "fun!"][-1023:1023]`: `["Python", "is", "very", "fun!"]`
8. `["Python", "is", "very", "fun!"][1023:-1023]`: `[]`
9. `["Python", "is", "very", "fun!"][1023:-1023:-1]`: `["fun!", "very", "is", "Python"]`

Exercise: Slicing Sequences for Retrieval

Evaluate the following slicing expressions.

10. `["Python", "is", "very", "fun!"][::0]`: `ValueError: slice step cannot be zero`
11. `"Python is very fun!"[::-1]`: `"!nuf yrev si nohtyP"`
12. `range(1023)[:1021]`: `range(1021)` or `range(0, 1021)`

iPRS Question 2

What is the output of the following program?

```
my_list = ["COMP", "1023", "is a", "good", "course"]
print(my_list[0][::], my_list[1][:500:1], sep=my_list[2][2])
```

A. `COMP␣`

B. `COMP 1023`

C. A runtime error occurs because of invalid **indexing**

D. A runtime error occurs because of invalid **slicing**

Note: ␣ denotes a space character for option A

iPRS Question 2 - Answer

What is the output of the following program?

```
my_list = ["COMP", "1023", "is a", "good", "course"]
print(my_list[0][::], my_list[1][:500:1], sep=my_list[2][2])
```

A. `COMP␣`

B. `COMP 1023`

C. A runtime error occurs because of invalid **indexing**

D. A runtime error occurs because of invalid **slicing**

Explanation

`my_list[0][::]`	`"COMP"`
`my_list[1][:500:1]`	`"1023"`
`my_list[2][2]`	A space character

The output would be `COMP` and 1023 with a space character in between, i.e., option B.

Slicing Sequences for Assignment

- When used as part of an assignment statement, it replaces the collection of elements covered by slicing with another sequence of elements on the right hand side (if the left-hand-side sequence is mutable)
  - The type of the sequence on the right hand side only needs to be iterable
- For this to work in all cases, the number of elements covered by slicing must be equal the number of elements in the sequence on the right hand side
  - Each element in the slice is replaced by the corresponding element of the sequence on the right hand side
  - This always works since the number of elements is equal
  - Somewhat intuitive...
- However, if `step` is 1, which is the default...
  - The number of elements does not need to be equal
  - All elements covered by the slice are removed
  - The new elements from the right-hand-side sequence are inserted at the `start` index
- Similar to slicing retrieval, out-of-range indices never trigger an IndexError

Exercise: Slicing Sequences for Assignment

State the value of `xx` after running each of the following code. Consider each case separately.

`xx` is initialized as `xx = ["Python", "is", "very", "fun!"]`

1. `xx = ["COMP", 1023]`
2. `xx[0] = ["COMP", 1023]`
3. `xx[1:3] = ["COMP", 1023]`
4. `xx[1:2] = ["COMP", 1023]`
5. `xx[1:1] = ["COMP", 1023]`
6. `xx[1:1] = []`
7. `xx[:] = []`
8. (demo) `xx[::2] = ["COMP", 1023]`
9. `xx[::3] = ["COMP", 1023]`
10. `xx[::4] = ["COMP", 1023]`
11. `xx[::-2] = ["COMP", 1023]`
12. `xx[::-1] = []`
13. (demo) `xx[::] = "COMP 1023"`

Exercise: Slicing Sequences for Assignment

State the value of `xx` after running each of the following code. Consider each case separately. `xx` is initialized as `xx = ["Python", "is", "very", "fun!"]`

1. `xx = ["COMP", 1023]`: `["COMP", 1023]`
   - This is simply assignment to a variable.
2. `xx[0] = ["COMP", 1023]`: `[["COMP", 1023], "is", "very", "fun!"]`
   - This is simply replacing an element of the `list` via indexing, not slicing it.
3. `xx[1:3] = ["COMP", 1023]`: `["Python", "COMP", 1023, "fun!"]`
4. `xx[1:2] = ["COMP", 1023]`: `["Python", "COMP", 1023, "very", "fun!"]`
   - When `step` is 1 (which is the case by default), the slice length does not need to be equal the iterable length on the right hand side
5. `xx[1:1] = ["COMP", 1023]`: `["Python", "COMP", 1023, "is", "very", "fun!"]`
6. `xx[1:1] = []`: `["Python", "is", "very", "fun!"]`
7. `xx[:] = []`: `[]`
   - One possible way to clear a `list`
   - Or replace the contents of a `list` if the iterable on the right hand side is nonempty

Exercise: Slicing Sequences for Assignment

State the value of `xx` after running each of the following code. Consider each case separately. `xx` is initialized as `xx = ["Python", "is", "very", "fun!"]`

8. `xx[::2] = ["COMP", 1023]`: `["COMP", "is", 1023, "fun!"]`
9. `xx[::3] = ["COMP", 1023]`: `["COMP", "is", "very", 1023]`
10. `xx[::4] = ["COMP", 1023]`: ValueError: attempt to assign sequence of size 2 to extended slice of size 1
    - When `step` is not 1, the slice length needs to equal the iterable length on the right hand side
11. `xx[::-2] = ["COMP", 1023]`: `["Python", 1023, "very", "COMP"]`
12. `xx[::-1] = []`: ValueError: attempt to assign sequence of size 0 to extended slice of size 4
    - `step` being -1 is handled as in when `step` is not 1
13. `xx[::] = "COMP 1023"`: `["C", "O", "M", "P", " ", "1", "0", "2", "3"]`
    - Any iterable can be used on the right hand side in slicing assignment
    - Iterables include `tuples`, `lists`, `strs`, `ranges`, ...

iPRS Question 3

Which of these lines leads to a runtime error?

```
1 my_list = ["COMP", "1023", "is a", "good", "course"]
2 my_list[2::3] = ["is definitely a"]
3 my_list[2][-2:] = ["not a"]
4 my_list[:1] = list(my_list[0])
```

A. Line 1

B. Line 2

C. Line 3

D. Line 4

iPRS Question 3 - Answer

Which of these lines leads to a runtime error?

```
1 my_list = ["COMP", "1023", "is a", "good", "course"]
2 my_list[2::3] = ["is definitely a"]
3 my_list[2][-2:] = ["not a"]
4 my_list[:1] = list(my_list[0])
```

A. Line 1

B. Line 2

C. Line 3

D. Line 4

Explanation

Strings are immutable. You cannot assign values to a string object. This would lead to a `TypeError`.

Line 1 is a valid list creation while lines 3 and 4 are valid assignments via slicing.

Another question: What would `print(my_list)` give if line 3 is removed?

Operators for Collections

- Equality (==, !=) is defined for collections
  - Two `lists`/`tuples` are equal if they are of the same type and they contain the same elements (using value equality) in the same order
  - Two `dicts` are equal if they contain the same key–value pairs (using value equality)
- Sequence concatenation
  - `"Roses" + " are red"` results in a `str` object `"Roses are red"`
  - `["Python"] + ["is", "fun"]` results in a `list` object `["Python", "is", "fun"]`
- Repeating sequences
  - `"ha" * 4` results in a `str` object `"hahahaha"`
  - `["ha", "he"] * 2` results in a `list` object `["ha", "he", "ha", "he"]`
- Remember these do not change/mutate the existing collections

Operators for Collections

- Previously, we have learnt lexicographic order for `strs`
- Essentially the same thing applies to sequences (ordered collections)
  - Syntax: `left_seq < right_seq` (similarly for `>`, `<=`, and `>=`)
  - Compare each element of the two sequences side-by-side
  - The comparison operators are used on each element
    - If you use the operator `<` to compare two sequences, the same operator `<` is used to compare each element in the sequences
    - If the operator is not supported between the two elements, a `TypeError` will be raised, e.g. comparing a `str` to an `int`
    - This may involve recursive lexicographic comparisons, e.g. sequences in sequences
  - The first element where the two sequences differ determines the order
    - Once this element is found, no more further comparison occurs, similar to short-circuiting logical expressions
  - If a sequence (e.g. `["Desmond"]`) is a prefix of another sequence (e.g. `["Desmond", "is", "good"]`), then the shorter sequence is lesser
  - You cannot compare sequences with different types, e.g. comparing a `list` with a `tuple`

Exercise: Operators for Collections

Evaluate the following expressions involving operators acting on collections.

1. `["Python", "is", "fun"] == ["Python", "is", "not", "fun"]`
2. `["Python", "is", "fun"] < ["Bython", "is", "fun"]`
3. `["COMP", 1021] < ["COMP", "1023"]`
4. `["COMP", 1021] < ["comp", "1023"]`
5. `[["COMP"]] < ["COMP"]`
6. `["COMP", ["A", "B"]] < ["COMP", ["B", "C"]]`
7. `["COMP1023", ["COMP1023"]] < ["COMP2011", "COMP 1023"]`
8. `["COMP1023", ["COMP1023"]] == ["COMP2011", "COMP 1023"]`
9. `["Python", "is", "fun"] == ("Python", "is", "fun")`

Exercise: Operators for Collections

Evaluate the following expressions involving operators acting on collections.

1. `["Python", "is", "fun"] == ["Python", "is", "not", "fun"]`: `False`
2. `["Python", "is", "fun"] < ["Bython", "is", "fun"]`: `False`
3. `["COMP", 1021] < ["COMP", "1023"]`: `TypeError: '<' not supported between instances of 'int' and 'str'`
   - Comparing the first element of both sequences, they are equal, so continue comparing
   - Comparing the second element runs `1021 < "1023"`, which is not supported
4. `["COMP", 1021] < ["comp", "1023"]`: `True`
   - The two sequences already differ in the first element, so comparison stops after the first element
   - That the second element of both sequences are not comparable has no effect
5. `[["COMP"]] < ["COMP"]`: `TypeError: '<' not supported between instances of 'list' and 'str'`
6. `["COMP", ["A", "B"]] < ["COMP", ["B", "C"]]`: `True`

Exercise: Operators for Collections

Evaluate the following expressions involving operators acting on collections.

7. `["COMP1023", ["COMP1023"]] < ["COMP2011", "COMP 1023"]`: `True`
   - Comparison is done element-by-element
   - The first element of both sequences are already different, so comparison stops after the them
   - Similar to Q3, `str` and `list` not being comparable has no effects
8. `["COMP1023", ["COMP1023"]] == ["COMP2011", "COMP 1023"]`: `False`
   - Every element is checked for equality
   - The second element of the two sequences differ in type
9. `["Python", "is", "fun"] == ("Python", "is", "fun")`: `False`
   - One is a `list`, the other is a `tuple`

Iterability

- An object is iterable if you can retrieve elements from it one at a time
- Refer to lecture slides
  - Traversing elements in a `list`
  - Traversing elements in a `tuple` (to be taught)
  - Traversing keys/pairs in a `dict` (to be taught)
- You can use `for`-loops or collection comprehension to process each element one-by-one
- Other iterable types
  - `range`: An immutable and ordered collection of `int`s
  - `str`: An immutable and ordered collection of individual characters

Copying Collections

- When we copy collections, generally there are two types of copying:
  - Shallow copying
  - Deep copying
- Why do we distinguish them?
  - Consider types that we have learnt before collections: `float`, `int`, `str`, ...
    - They are all immutable
  - What if an element in a collection is a mutable object?
  - Objects contained in a collection can be considered as "subobjects" of the collection
  - You can copy the collection itself only and "reuse" the "subobjects"
    - This is what we call shallow copying
  - You can copy the collection and all the "subobjects" within, and all the "subobjects" within "subobjects", and all the "subobjects" within "subobjects" within "subobjects"...
    - This is what we call deep copying
    - We don't have to deep copy everything
    - You can stop at a certain level

Copying Collections

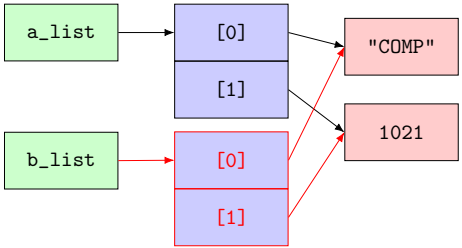
- Actually, the above concept also applies to any object with "subobjects" inside, not just collections
- The copying functions below will also work for most other types, not just collections
- We will consider collections only for simplicity for now
- Import the `copy` module to use copying functions
  - `import copy`
  - `from copy import copy, deepcopy`
- The examples on the next few pages show the difference between shallow and deep copying:
  - Note: The following examples assume the `str` object `"COMP"` and the `int` objects `1021` and `1023` are cached.

Shallow Copying Collections

An example where elements are all immutable:

```
1  import copy
2  a_list = ["COMP", 1021]
3  b_list = copy.copy(a_list)
4  # Same as `b_list = list(a_list)`
5  print(a_list is b_list) # False
6
7  a_list[1] = 1023
8  print(b_list)           # ['COMP', 1021]
```

After line 3:

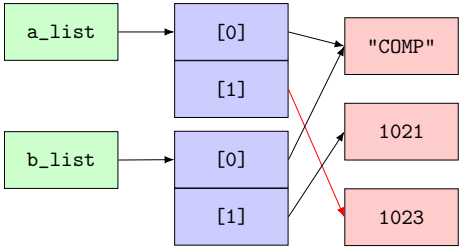


Shallow Copying Collections

An example where elements are all immutable:

```
1  import copy
2  a_list = ["COMP", 1021]
3  b_list = copy.copy(a_list)
4  # Same as `b_list = list(a_list)`
5  print(a_list is b_list) # False
6
7  a_list[1] = 1023
8  print(b_list)            # ['COMP', 1021]
```

After line 7:

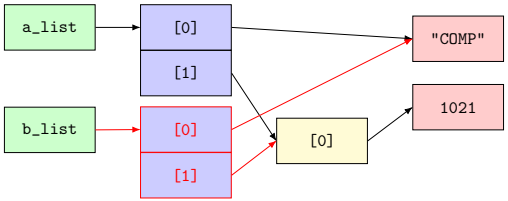


Shallow Copying Collections (Cont.)

An example where an element (the `list`) is mutable:

```
1  import copy
2  a_list = ["COMP", [1021]]
3  b_list = copy.copy(a_list)
4  # Same as b_list = list(a_list)
5  print(a_list is b_list)        # False
6  print(a_list[1] is b_list[1]) # True
7
8  a_list[1][0] = 1023
9  print(b_list) # ['COMP', [1023]]
10
11 a_list[1] = [1021]
12 print(b_list) # ['COMP', [1023]]
```

After line 3:

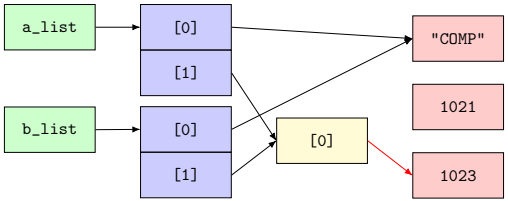


Shallow Copying Collections (Cont.)

An example where an element (the `list`) is mutable:

```
1  import copy
2  a_list = ["COMP", [1021]]
3  b_list = copy.copy(a_list)
4  # Same as b_list = list(a_list)
5  print(a_list is b_list)        # False
6  print(a_list[1] is b_list[1]) # True
7
8  a_list[1][0] = 1023
9  print(b_list) # ['COMP', [1023]]
10
11 a_list[1] = [1021]
12 print(b_list) # ['COMP', [1023]]
```

After line 8:

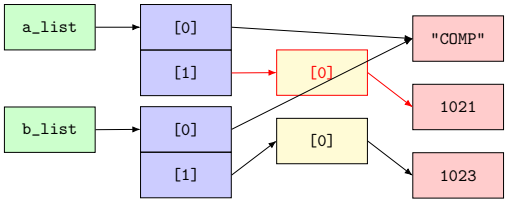


Shallow Copying Collections (Cont.)

An example where an element (the `list`) is mutable:

```
1  import copy
2  a_list = ["COMP", [1021]]
3  b_list = copy.copy(a_list)
4  # Same as b_list = list(a_list)
5  print(a_list is b_list)        # False
6  print(a_list[1] is b_list[1]) # True
7
8  a_list[1][0] = 1023
9  print(b_list) # ['COMP', [1023]]
10
11 a_list[1] = [1021]
12 print(b_list) # ['COMP', [1023]]
```

After line 11:

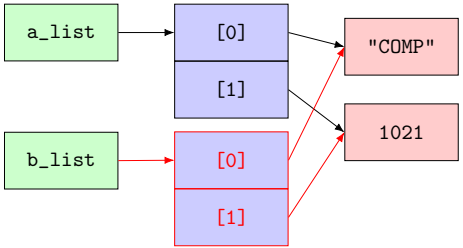


Deep Copying Collections

An example where elements are all immutable:

```
1  import copy
2  a_list = ["COMP", 1021]
3  b_list = copy.deepcopy(a_list)
4  print(a_list is b_list) # False
5
6  a_list[1] = 1023
7  print(b_list) # ['COMP', 1021]
```

After line 3:

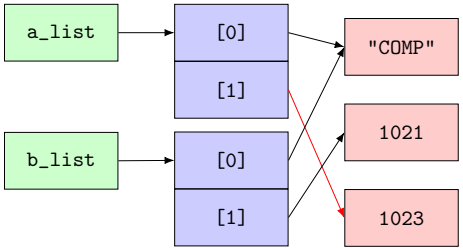


Deep Copying Collections

An example where elements are all immutable:

```
1  import copy
2  a_list = ["COMP", 1021]
3  b_list = copy.deepcopy(a_list)
4  print(a_list is b_list) # False
5
6  a_list[1] = 1023
7  print(b_list) # ['COMP', 1021]
```

After line 6:



It's... the same as shallow copy?

Indeed! Since every element in `a_list` is immutable, nor are there nested collections, there are no differences!

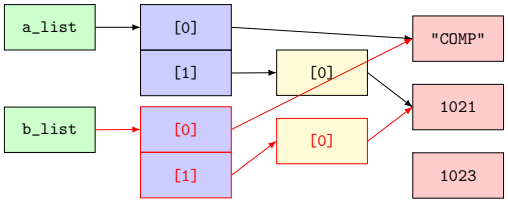
Our next example will show what happens when an element is mutable.

Deep Copying Collections (Cont.)

An example where an element (the `list`) is mutable:

```
1  import copy
2  a_list = ["COMP", [1021]]
3  b_list = copy.deepcopy(a_list)
4  print(a_list is b_list)        # False
5  print(a_list[1] is b_list[1]) # False
6
7  a_list[1][0] = 1023
8  print(b_list) # ['COMP', [1021]]
9
10 a_list[1] = [1021]
11 print(b_list) # ['COMP', [1021]]
```

After line 3:

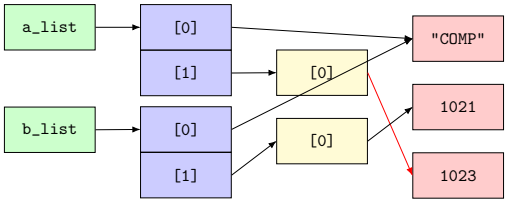


Deep Copying Collections (Cont.)

An example where an element (the `list`) is mutable:

```
1  import copy
2  a_list = ["COMP", [1021]]
3  b_list = copy.deepcopy(a_list)
4  print(a_list is b_list)        # False
5  print(a_list[1] is b_list[1]) # False
6
7  a_list[1][0] = 1023
8  print(b_list) # ['COMP', [1021]]
9
10 a_list[1] = [1021]
11 print(b_list) # ['COMP', [1021]]
```

After line 7:

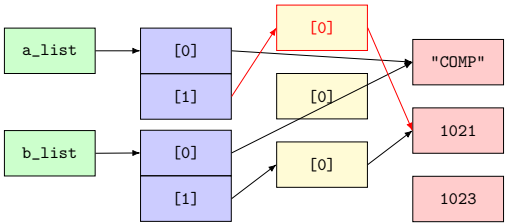


Deep Copying Collections (Cont.)

An example where an element (the `list`) is mutable:

```
1  import copy
2  a_list = ["COMP", [1021]]
3  b_list = copy.deepcopy(a_list)
4  print(a_list is b_list)        # False
5  print(a_list[1] is b_list[1]) # False
6
7  a_list[1][0] = 1023
8  print(b_list) # ['COMP', [1021]]
9
10 a_list[1] = [1021]
11 print(b_list) # ['COMP', [1021]]
```

After line 10:



Exercise: Copying Collections

Determine if a copy of the `list` object `xx` is made in each of the following expressions.

1. `yy = xx`
2. `xx[0] = xx`
3. `xx[:] = xx`
4. `yy = list(xx)`
5. `yy = xx[:]`
6. `yy = [] + xx`
7. `yy = copy.copy(xx)`
8. `yy = copy.deepcopy(xx)`
9. `yy = copy.copy([xx])`
10. `yy = copy.deepcopy([xx])`
11. `yy = [yy for yy in xx]` (Hint: This uses `list comprehension`.)

Exercise: Copying Collections

Determine if a copy of the `list` object `xx` is made in each of the following expressions.

1. `yy = xx`: No
2. `xx[0] = xx`: No
3. `xx[:] = xx`: No
4. `yy = list(xx)`: Yes
5. `yy = xx[:]`: Yes
6. `yy = [] + xx`: Yes
7. `yy = copy.copy(xx)`: Yes
8. `yy = copy.deepcopy(xx)`: Yes
9. `yy = copy.copy([xx])`: No
10. `yy = copy.deepcopy([xx])`: Yes
11. `yy = [yy for yy in xx]` (Hint: This uses `list comprehension`.): Yes

Collection Comprehension

- Given a collection, you may want to perform an action and get a result for each element of the collection, and gather the results as a new collection (map)
  - The below code maps each element of `a_list` to a number that is twice the element to create `b_list`
  - ```
    b_list = []
    for ele in a_list:
        b_list.append(ele * 2)
    ```
  - (Almost) equivalent one-line code: `b_list = [ele * 2 for ele in a_list]`
- Given a collection, you may want to keep or discard certain elements of the collection to create a new collection (filter)
  - The below code filters for (keeps) elements of `a_list` that are nonnegative to create `b_list`
  - ```
    b_list = []
    for ele in a_list:
        if ele >= 0:
            b_list.append(ele)
    ```
  - (Almost) equivalent one-line code: `b_list = [ele for ele in a_list if ele >= 0]`

List Comprehension

- A good way to understand `list` comprehension is to start with the multiple-line code in the previous slides
- Given the following code creating a `list`:

```
b_list = []
for ele in a_list:
    b_list.append(ele * 2)
c_list = []
for ele in a_list:
    if ele >= 0:
        c_list.append(ele)
```

- These can be written using 2 `list` comprehensions:

```
b_list = [ele * 2 for ele in a_list]
c_list = [ele for ele in a_list if ele >= 0]
```

- Can you observe any patterns?

List Comprehension

- Original multiple-line code:

```
c_list = []
for ele in a_list:          # line 1
    if ele >= 0:            # line 2
        c_list.append(ele) # line 3
```

- `list` comprehension (in multiple lines for clarity):

```
c_list = [ele               # line 3
    for ele in a_list       # line 1
    if ele >= 0]            # line 2
```

- Comprehension transforms and then puts the last line appending to the collection as the first line
- In fact, this pattern holds for any arbitrary order of nested `for` and `if`, given the first line is a `for`-statement

Exercise: List Comprehension

Evaluate or describe (if the output would be too long) the following expressions involving comprehension.

1. `[abc for abc in a_list]`, where `a_list` is a `list` object
2. `[xyz * 2 for xyz in range(10) if xyz % 2 == 1]`
3. `[char for char in "COMP 1023" if char != " "]`
4. `[[xx, yy] for xx in range(10) for yy in range(10)]`
5. `[[xx, yy] for xx in range(10) for yy in range(10) if yy % 3 == 0]`
6. `[[xx, yy] for xx in range(10) if yy % 3 == 0 for yy in range(10)]`

Exercise: List Comprehension

Evaluate or describe (if the output would be too long) the following expressions involving comprehension.

1. `[abc for abc in a_list]`, where `a_list` is a `list` object: Shallow-copies `a_list`.
2. `[xyz * 2 for xyz in range(10) if xyz % 2 == 1]`: `[2, 6, 10, 14, 18]`
   - You can find the output by following the items below one-by-one
   - `[xyz for xyz in range(10)]`: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
   - `[xyz for xyz in range(10) if xyz % 2 == 1]`: `[1, 3, 5, 7, 9]`
   - `[xyz * 2 for xyz in range(10) if xyz % 2 == 1]`: `[2, 6, 10, 14, 18]`
3. `[char for char in "COMP 1023" if char != " "]`: `['C', 'O', 'M', 'P', '1', '0', '2', '3']`

Exercise: List Comprehension

Evaluate or describe (if the output would be too long) the following expressions involving comprehension.

4. `[[xx, yy] for xx in range(10) for yy in range(10)]`: A `list` of pairs (stored as `lists`) of integers, where both numbers run from 0 to 9 (both inclusive).
   - Equivalent multiple line code:

```
result = []
for xx in range(10):
    for yy in range(10):
        result.append([xx, yy])
print(result)
```

5. `[[xx, yy] for xx in range(10) for yy in range(10) if yy % 3 == 0]`: Similar to the previous item, but the right integer in each pair is divisible by 3.
   - Equivalent multiple line code:

```
result = []
for xx in range(10):
    for yy in range(10):
        if yy % 3 == 0:
            result.append([xx, yy])
print(result)
```

Exercise: List Comprehension

Evaluate or describe (if the output would be too long) the following expressions involving comprehension.

6. `[[xx, yy] for xx in range(10) if yy % 3 == 0 for yy in range(10)]`:
   UnboundLocalError: cannot access local variable 'yy' where it is not associated with a value
   - Equivalent multiple line code:

```
result = []
for xx in range(10):
    if yy % 3 == 0:              # Where should `yy` come from?
        for yy in range(10):
            result.append([xx, yy])
print(result)
```

iPRS Question 4

What is the output of the following program?

```
my_list1 = [i for i in range(2, 7)]
my_list2 = [[j + 1] for j in my_list1]
print(my_list1 < my_list2)
```

A. `True`

B. `False`

C. A runtime error occurs

iPRS Question 4 - Answer

What is the output of the following program?

```
my_list1 = [i for i in range(2, 7)]
my_list2 = [[j + 1] for j in my_list1]
print(my_list1 < my_list2)
```

A. `True`
B. `False`
C. A runtime error occurs

Explanation

`my_list1`	`[2, 3, 4, 5, 6]`
`my_list2`	`[[3], [4], [5], [6], [7]]`

When comparing these 2 lists, the first elements, 2 and `[3]` are compared. Since comparison between integers and lists are not supported, a runtime error occurs.
Another question: If there is `my_list3 = [j for j in my_list1]`, would `my_list3` be a shallow or deep copy of `my_list1`?

Useful technique: `str` splitting

- If the separator is provided
  - The separator cannot be an empty `str`
  - Each occurrence of the separator is used to split the `str` once
    - Consuective occurrences are not treated specially
  - `"abc".split("b")`: `["a", "c"]`
  - `"abbc".split("b")`: `["a", "", "c"]`
  - `"".split("b")`: `[""]`
- If the separator is omitted
  - Then the separator is any whitespace, which includes space and newline; in general, it is any character that is invisible by itself
  - Each consecutive run of whitespaces is used to split the `str` once
  - Any empty `str`s after splitting are discarded
  - `"a c".split()`: `["a", "c"]`
  - `"a  c".split()`: `["a", "c"]`
  - `"a\nc".split()`: `["a", "c"]`
    - `\n` represents a newline
  - `"".split()`: `[]`

Exercise: `str` splitting

Evaluate the following expressions using `str.split`.

1. `"Python is fun!".split(" ")`
2. `"Python is fun!".split("  ")`
3. `"Python is fun!".split()`
4. `"Python  is  fun!".split(" ")` (double spaces)
5. `"Python  is  fun!".split()` (double spaces)
6. `"".split(" ")`
7. `"".split()`
8. `"Python is fun!".split("")`

Exercise: `str` splitting

Evaluate the following expressions using `str.split`.

1. `"Python is fun!".split(" ")`: `["Python", "is", "fun!"]`
2. `"Python is fun!".split("  ")`: `["Python is fun!"]`
3. `"Python is fun!".split()`: `["Python", "is", "fun!"]`
4. `"Python  is  fun!".split(" ")` (double spaces): `["Python", "", "is", "", "fun!"]`
5. `"Python  is  fun!".split()` (double spaces): `["Python", "is", "fun!"]`
6. `"".split(" ")`: `[""]`
7. `"".split()`: `[]`
8. `"Python is fun!".split("")`: ValueError: empty separator

That's all!

Any question?

Appendix

- These materials are optional
- Feel free to read them if you are interested

Slicing Sequences

- Syntax: `sequence[start:stop:step]`, where `start`, `stop`, and `step` are `int`s
- `step` cannot be 0, otherwise a `ValueError` occurs
- The 2nd colon (`:`) is optional
- All 3 arguments are optional; their default values if omitted:
  - `step` defaults to 1
  - `start` defaults to 0 if `step` is positive, or -1 if `step` is negative
  - `stop` defaults to an index after the sequence end if `step` is positive, or an index before the sequence start if `step` is negative
  - Revisit these defaults later...
- Types that can be sliced: if a type can be indexed, it often can also be sliced
- Normally, retrieves a collection of elements from the sequence as a new sequence
  - Its type is the same as that of the original sequence: if you slice an `A`, you get an `A`; applies for `list`, `tuple`, `str`, `range`, ...
  - Negative indices have the same meaning as in indexing
  - The detailed steps are complicated and are given in the next slide

Slicing Sequences: Detailed Steps

1. Use default values for omitted arguments (`start`, `stop`, `step`)
2. Interpret `start` and `stop` as pointing to actual elements in the sequence
   - Any in range index can be written either as a nonnegative index or negative index; they are aliases of each other
   - If the index is before the sequence start: for `start`, it points to the first element; for `stop`, it points to an imaginary 'element' on the left of the first element
   - If the index is after the sequence end, for `start`: it points to the last element; for `stop`, it points to an imaginary 'element' on the right of the last element
   - In particular, the `stop` default is out of range, and thus always triggers the above two cases
3. Use the sign of `step` as the slicing direction
   - If positive, an index `x` is before another index `y` if `x` points to an element on the left of that pointed to by `y`
   - If negative, an index `x` is before another index `y` if `x` points to an element on the right of that pointed to by `y`
   - The actual numeric value of the indices `x` and `y` do not matter; that is, an index being negative does not imply it is always before any nonnegative index for positive `step`
4. (Continue...)

Slicing Sequences: Detailed Steps

4. Check if `start` is before `end`; if not, the slice is empty
5. Otherwise, the element at the `start` index is always included; let this be the current index
6. Repeatedly add a new element to the slice or stop
   - Add `step` (including its sign) to the current index and check if the current index is before the `stop` index
   - If yes, append the element at the current index to the slice, and repeat this step
   - If no, the element at the current index is not appended to the slice, and we stop

Slicing Sequences

- Some examples to help understand and consolidate the complicated detailed steps
- We use the following `tuple` created by:
  `seq = "Isn't", "Python", "fun?",`
- We visualize `seq[start:stop:step]` (e.g. `seq[0:2:1]`) like this:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
	→ `start`	→	`stop`	

  - Elements pointed to by `start` and `stop` are labeled
  - Elements included in the slice are labeled by an arrow pointing into the `step` direction: positive is right, negative is left

Slicing Sequences

- Consider the simplest cases: assume `step` is 1
- The following 4 statements are all equivalent and produces `["Isn't", 'Python']`
  - `seq[0:2]`
  - `seq[-3:-1]`
  - `seq[0:-1]`
  - `seq[-3:2]`
- They are all represented by the same diagram:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
	→ `start`	→	`stop`	

Slicing Sequences

- In slicing (and indexing), nonnegative indices and negative indices are aliases of each other
- Each in-range element can be indexed using two numbers: a nonnegative index counting from sequence start, a negative index counting from sequence end
- The actual numeric values do not matter
  - That 0 is numerically larger than -1 does not make `seq[0:-1]` an empty slice
- You can replace one alias by the other alias in your program without any changes in the program behavior
- This holds for other values of `step` as well
- Thus, our diagrams put two numbers above each element

Slicing Sequences

- (Consider the simplest cases: assume `step` is 1)
- The default for `start` is 0, i.e. the first element; this is easy to understand
- The default for `stop` is an index after the sequence end
  - Seems very unintuitve at first
  - Implies the default `stop` points to the 'imaginary' element on the right of sequence end
- See next slide for two examples

Slicing Sequences

- (Consider the simplest cases: assume `step` is 1)
- Diagram for `seq[0:]`:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
	→ `start`	→	→	`stop`

- Diagram for `seq[:-1]`:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
	→ `start`	→	`stop`	

- The rationale for the different defaults of `start` and `end` is because slicing includes the start element but excludes the stop element
  - A few other programming languages includes both the start and stop elements, e.g. MATLAB

Slicing Sequences

- Consider more complicated cases: assume `step` is -1
- As an exercise to the reader: guess the behavior and check with later slides

Slicing Sequences

- (Consider more complicated cases: assume `step` is -1)
- The following 4 statements are all equivalent and produces `["fun?", 'Python']`
  - `seq[2:0:-1]`
  - `seq[-1:-3:-1]`
  - `seq[2:-3:-1]`
  - `seq[-1:0:-1]`
- They are all represented by the same diagram:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
	`stop`	←	← `start`	

- Notice now the element pointed to by `stop` is on the left of that by `start`; otherwise, the slice would have been empty

Slicing Sequences

- (Consider more complicated cases: assume `step` is -1)
- The default for `start` is -1, i.e. the last element; this is easy to understand
- The default for `stop` is an index before the sequence start
  - Also seems very unintuitve at first
  - Implies the default `stop` points to the 'imaginary' element on the left of sequence start
- See next slide for two examples

Slicing Sequences

- (Consider more complicated cases: assume `step` is -1)
- Diagram for `seq[-1::-1]`:

⋯	0/−3	1/−2	2/−1	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
`stop`	←	←	← `start`	

- Diagram for `seq[:0:-1]`:

⋯	0/−3	1/−2	2/−1	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
	`stop`	←	← `start`	

- The rationale for the different defaults of `start` and `end` is because slicing includes the start element but excludes the stop element

Slicing Sequences

- It would be interesting to write the default `start` and `stop` indices explicitly instead of omitting them...
- `seq[:x]` is equivalent to `seq[0:x]`; nothing interesting
- `seq[x:]` is equivalent to `seq[x:len(seq)]`
  - `len(seq)` represents an index after the last element
  - Any integer that is at least `len(seq)` would also work
- `seq[:x:-1]` is equivalent to `seq[-1:x:-1]`; nothing interesting
- `seq[x::-1]` is equivalent to `seq[x:-len(seq)-1:-1]`
  - `-len(seq)` is a negative index that indexes the first element
  - Then `-len(seq)-1` represents an index before the first element
  - `-1` would not work as it refers to the last element
  - Any integer not greater than `-len(seq)-1` would also work

Slicing Sequences

- Slicing can result in empty sequences
- Diagram for `seq[2:1]`:

⋯	0/−3	1/−2	2/−1	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
		`stop`	`start`	

- Diagram for `seq[1:2:-1]`:

⋯	0/−3	1/−2	2/−1	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
		`start`	`stop`	

- The `step` direction matters!

Slicing Sequences

- If `start` and `stop` refer to the same element, slicing also result in empty sequences
- Diagram for `seq[1:1]`:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
		`start`, `stop`		

- Diagram for `seq[1:1:-1]`:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
		`start`, `stop`		

Slicing Sequences

- Finally, consider the most general cases: `step` is any nonzero integer
- Diagram for `seq[:2:2]`:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
	→ `start`		`stop`	

- Diagram for `seq[::2]`:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
	→ `start`		→	`stop`

- Python tries to fit as many elements (arrows) as possible

Slicing Sequences

- (Finally, consider the most general cases: `step` is any nonzero integer)
- Diagram for `seq[:0:-2]`:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
	`stop`		← `start`	

- Diagram for `seq[::-2]`:

⋯	**0/−3**	**1/−2**	**2/−1**	⋯
(imaginary)	`"Isn't"`	`"Python"`	`"fun?"`	(imaginary)
`stop`	←		← `start`	

Slicing Sequences

- Unlike indexing, out-of-range indices are handled gracefully, and never trigger an IndexError
- Refer to the detailed steps above
- These expressions are equivalent and copy `seq`
  - `seq[:]`
  - `seq[:len(seq)]`
  - `seq[:len(seq)+1]`
  - `seq[:len(seq)+2]`
  - ...
  - (separator)
  - `seq[:]`
  - `seq[0:]`
  - `seq[-len(seq):]`
  - `seq[-len(seq)-1:]`
  - `seq[-len(seq)-2:]`
  - ...

Slicing Sequences

- These expressions are equivalent and produce an empty sequence
  - `seq[len(seq):]`
  - `seq[len(seq)+1:]`
  - `seq[len(seq)+2:]`
  - ...
  - (separator)
  - `seq[:0]`
  - `seq[:-len(seq)]`
  - `seq[:-len(seq)-1]`
  - `seq[:-len(seq)-2]`
  - ...

Slicing Sequences

- These expressions are equivalent and copy `seq` in reverse
  - `seq[::-1]`
  - `seq[len(seq)-1::-1]`
  - `seq[len(seq)::-1]`
  - `seq[len(seq)+1::-1]`
  - ...
  - (separator)
  - `seq[::-1]`
  - `seq[:-len(seq)-1:-1]`
  - `seq[:-len(seq)-2:-1]`
  - `seq[:-len(seq)-3::-1]`
  - ...

Slicing Sequences

- These expressions are equivalent and produce an empty sequence
  - `seq[:len(seq)-1:-1]`
  - `seq[:len(seq):-1]`
  - `seq[:len(seq)+1:-1]`
  - ...
  - (separator)
  - `seq[-len(seq)-1::-1]`
  - `seq[-len(seq)-2::-1]`
  - ...