# Practical Programming
## *in Python*

*Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo*

## Lecture 8
## *Lists*

*Storing collections of data, Mutability, Using Lists*

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam International Centre for Theoretical Physics

# Abstract

*"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."*
*– S. Kelly-Bootle*

We will see that it is next to impossible to write useful programs without storing collections of of data.
So far we have seen numbers, Boolean values, strings, functions and a few other types. Objects of these types, once created, can't be modified. Lists are introduces as the first example of a *mutable* type in Python.
The ability to change lists and the many operations and methods available for lists makes them very powerful.
You will use a lot of lists in your programs.

# Overview

- Why we need to store collections of data
- Lists as the first example of a container for storing collections
- Mutability: lists can be changed
- Operations on lists
- Slicing
- List methods

**Lists are the most common collection type in Python.**

# Example: Counting Whales

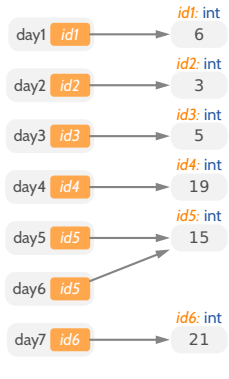
- The data is taken from the *Gray Whales Count 2016*
- Day 1 is February 16, 2016
- The last observation day in 2016, day 101, was May 26

| Day | Whales Sighted |
|---|---|
| 1 | 6 |
| 2 | 3 |
| 3 | 5 |
| 4 | 19 |
| 5 | 15 |
| 6 | 15 |
| 7 | 21 |
| … | |

**The full table has 101 rows!**

# How Can We Represent this Data Set?

- Given what we have seen so far, we would have to create seven variables for just this one week
- To track the whole observation period we would need 101 variables
- And explicitely reference each of them by name in our program




**This is a programming nightmare.**

# List Expressions

- The general form of a `list` expression is:

```
[expression1, expression2, ..., expressionN]
```

- The empty `list` is expressed as:

```
[]
```

- The `list` itself is an object
- It contains *items* or *elements*
- The items contain the memory addresses of other objects
- The items are ordered an can be accessed via *indices*

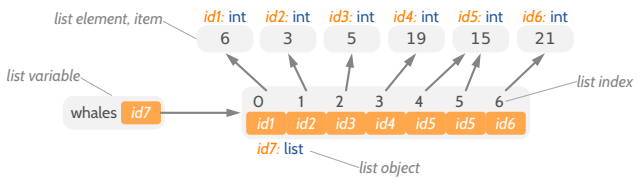
**The indices start at `0` not `1`! We will motivate this choice soon.**

# List Variables

- We can use a `list` to keep track of the one week (or a whole year!) of whale counts
- That is, we can use a `list` to keep track of the seven `int` objects containing the counts:

```
>>> whales = [6, 3, 5, 19, 15, 15, 21]
>>> whales
[6, 3, 5, 19, 15, 15, 21]
```

- A list is an object that can be assigned to a variable:




**We can now refer to a collection by a name.**

# Using Indices: Referring to List Items

- To refer to an item we put the index in brackets after a reference to a list:

```
>>> whales = [6, 3, 5, 19, 15, 15, 21]
>>> whales[0]
6
>>> whales[1]
3
>>> whales[5]
15
>>> whales[6]
21
```

- Using a list index that is *out of range* results in an error:

```
>>> whales = [6, 3, 5, 19, 15, 15, 21]
>>> whales[999]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

**Because indices start from `0` the highest valid index is `N-1`.**

# Using Indices: Referring to List Items

- Python allows us to use negative indices as well:

```
>>> whales = [6, 3, 5, 19, 15, 15, 21]
>>> whales[-1]
21
>>> whales[-2]
15
>>> whales[-7]
6
```

- Negative indices can also be out of range:

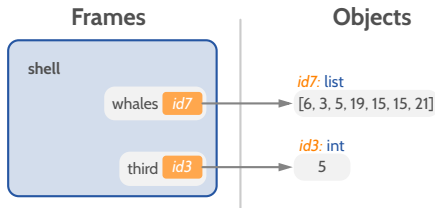
```
>>> whales = [6, 3, 5, 19, 15, 15, 21]
>>> whales[-17]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

**Negative indices are useful because we don't need to know the length of the list.**

# Variables Referring to List Items

- Each item is a reference to an object
- That means we can assign it to a variable:

```
>>> whales = [6, 3, 5, 19, 15, 15, 21]
>>> third = whales[2]
>>> print('Third day:', third)
Third day: 5
```




**The concept of aliasing becomes very important with lists.**

# The Empty List

- We have learned about the *empty string* in Lecture 4
- There is also an *empty list*
- The empty list (like all lists) is expressed with brackets:

```
>>> whales = []
```

- An empty list has no items
- So any attempt to index it results in an error:

```
>>> whales[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> whales[-1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

**In a Boolean expressions the empty list evaluates to `False`.**

# Lists Are Heterogeneous

- Python lists can contain any kind of data
- In particular, items can have different types:

```
>>> krypton = ['Krypton', 'Kr', -157.2, -153.4]
>>> krypton[1]
'Kr'
>>> krypton[2]
-157.2
```

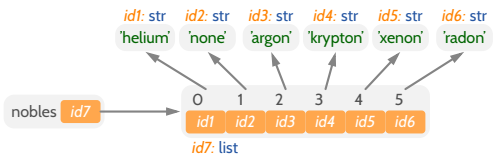
**This is very useful. In practice, Python list items tend to have the same type.**

# Modifying Lists

- Suppose you made a typo when defining a list of noble gases:

```
>>> nobles = ['helium', 'none', 'argon', 'krypton', 'xenon', 'radon']
```

- This is the result in the memory model:



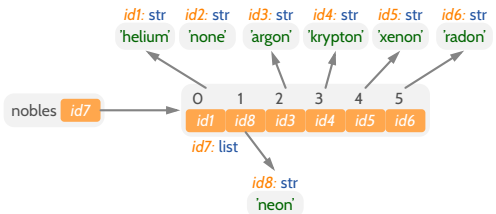

**Collections can be large; it would not be practical to redefine them as a whole.**

# Modifying Lists

- Python allows us to *mutate* the list, that is, change the list's contents:

```
>>> nobles[1] = 'neon'
>>> nobles
['helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']
```

- This is the result in the memory model:




**We say lists are *mutable* (can be changed/modified/*mutated*).**

# Mutable vs. Immutable

- List are *mutable*
- That is, an expression like `L[i]` can appear on the left hand side of an assignment
- This means "look up the memory address at index `i` so it can be overwritten":

```
>>> nobles[1] = 'neon'
```

- Strings are *immutable*
- An expression like `S[i]` can *not* appear on the left hand side of an expression
- Some methods appear to change strings, but in fact they do create new strings:

```
>>> name = 'Darwin'
>>> capitalized = name.upper()
>>> print(capitalized, id(capitalized))
DARWIN 140471607498096
>>> print(name, id(name))
Darwin 140471607498040
```

**Understanding mutability is *very* important for Python programmers.**

# List Functions

- We have already seen plenty of Python's built-in functions
- Some, like `len()`, work on lists
- There are more we haven't seen yet because they only make sense for collections

| Function | Description | Requirements |
|---|---|---|
| `len(L)` | Return the number of items in list `L` | None |
| `max(L)` | Return the maximum value in list `L` | Items can be compared |
| `min(L)` | Return the minimum value in list `L` | Items can be compared |
| `sum(L)` | Return the sum of the values in list `L` | Items can be added |
| `sorted(L)` | Return a sorted *copy* of list `L` | Items can be compared |

**The above functions, *including* `sorted()`, do *not mutate* the list `L`.**

# List Functions: Examples

- Different isotopes of chemical elements can be stable or radioactive
- Radioactive isotopes often have different half-lives
- For instance, Pu-238, Pu-239, Pu-240, Pu-241, Pu-242

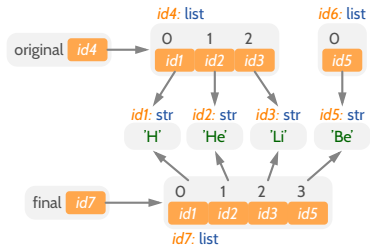
```
>>> half_lives = [887.7, 24100.0, 6563.0, 14.0, 373300.0]
>>> len(half_lives)
5
>>> max(half_lives)
373300.0
>>> min(half_lives)
14.0
>>> sum(half_lives)
404864.7
# creates a new list
[14.0, 887.7, 6563.0, 24100.0, 373300.0]
>>> half_lives
[887.7, 24100.0, 6563.0, 14.0, 373300.0]
```

**Note again that the list `half_lives` was *not changed*.**

# Operations on Lists

- Built-in Python types also provide operators when they make sense
- We have seen the usual examples for numbers
- We also have seen that **+** concatenates strings
- This operation makes sense for lists, too:

```
>>> original = ['H', 'He', 'Li']
>>> final = original + ['Be']
>>> final
['H', 'He', 'Li', 'Be']
>>> original
['H', 'He', 'Li']
```




**Note that this also *does not mutate* the original list – a new list is created.**

# The List Type

- A Python `list` is a type
- Python complains if you try to use functions or operators that are not defined for the involved types:

```
>>> ['H', 'He', 'Li'] + 'Be'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

**Python is *not* weakly typed.**

# List Operators: Repetition

- Lists can be repeated with the * operator:

```
>>> metals = ['Fe', 'Ni']
>>> metals * 3
['Fe', 'Ni', 'Fe', 'Ni', 'Fe', 'Ni']
```

**Please don't call this multiplication.**

# List Operators: Deletion

- Since lists are mutable, you can delete an item from a list
- This is done with the **`del`** operator:

```
>>> metals = ['Fe', 'Ni']
>>> del metals[0]
>>> metals
['Ni']
```

**Note that `del` takes an index, not a value.**

# List Operators: Containment

- We can use the **`in`** operator to determine containment:

```
>>> nobles = ['helium', 'neon', 'argon', 'krypton', 'xenon', 'radon']
>>> gas = input('Enter a gas: ')
Enter a gas: argon
>>> if gas in nobles:
...     print('{} is noble.'.format(gas))
...
argon is noble.
```

- Unlike with strings, operator **`in`** *can't* be used to check for sub-lists:

```
>>> [1, 2] in [0, 1, 2, 3]
False
```

**Why do you think Python allows to check for sub-strings but not sub-lists?**

# Example: C. elegans – "The Worm"

- C. elegans – "The Worm" – is a small worm, roughly ∼1 mm in size
- It is studied a lot in biology because it possesses many features of "higher" organisms
- You can learn more about it *here*
- Biologists label its appearances and behaviour (phenotypes) with three-letter codes
- Some of these labels are less useful because they are hard to distinguish in practice
- For example, "Dpy" and "Sma"

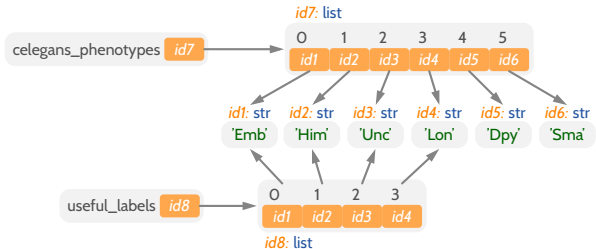
| Label | Phenotype |
|---|---|
| Emp | embryonic lethality |
| Him | high incidence of males |
| Unc | uncoordinated |
| Lon | long |
| Dpy | dumpy: short and fat |
| Sma | small |
| … | |

**Given a full list, how can we extract the more useful labels?**

# Slicing Lists

- Python has a convenient notation for taking a *slice* of a list: `L[i:j]`

```
>>> celegans_phenotypes = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_phenotypes
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> useful_labels = celegans_phenotypes[0:4] # take the first four elements
>>> useful_labels
['Emb', 'Him', 'Unc', 'Lon']
```

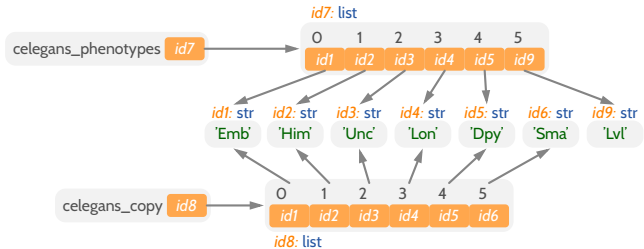



**Slicing a list *creates a new list*.**

# Cloning Lists

- Taking a whole slice *clones* (copies) a list: `L[:]`

```
>>> celegans_phenotypes = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_copy = celegans_phenotypes[:] # make a copy (clone) the list
>>> celegans_phenotypes[5] = 'Lvl' # this does not change (mutate) the clone!
>>> celegans_phenotypes
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> celegans_copy
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
```

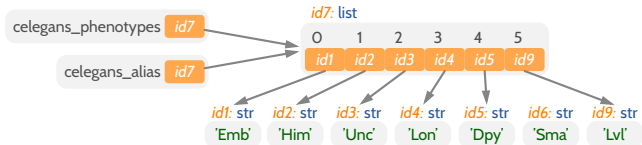



**Mutating the original does *not* change the clone.**

# Aliasing: What's in a Name?

- Aliasing is one of the reasons the notion of *mutability* is important
- A change to the original mutates the alias and vice versa

```
>>> celegans_phenotypes = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_alias = celegans_phenotypes # create an alias (a new name for the same object)
>>> celegans_phenotypes[5] = 'Lvl' # this does change (mutate) the alias!
>>> celegans_phenotypes
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> celegans_alias
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
```




**An *alias* is a name referring to the *same object* as another name.**

# Mutable Parameters

- Parameters are just variables
- If parameters are mutable, a function can change them:

```
def remove_last_item(L):
    """
    Return L with the last item removed.

    Precondition:

        len(L) > 0

    Examples:

        >>> remove_last_item([1, 3, 2, 4])
        [1, 3, 2]
    """
    del L[-1]
    return L
```

- Do we need to return `L` in this case to make the function useful?

**In general it is a bad idea to change mutable parameters in a function.**

# Common List Methods

| Method | Description |
|---|---|
| `L.append(v)` | Append the value `v` to the list `L` |
| `L.clear()` | Remove all items from list `L` |
| `L.count(v)` | Return the number of occurrences of `v` in `L` |
| `L.extend(l)` | Append the items in `l` to `L` |
| `L.index(v)` | Return the index of the first occurrence of `v` in `L` |
| `L.index(v, beg)` | Return the index of the first occurrence of `v` in `L` at or after the index `beg` |
| `L.index(v, beg, end)` | Return the index of the first occurrence of `v` between indices `beg` (inclusive) and `end` (exclusive) |
| `L.insert(i, v)` | Insert the value `v` at index `i`, shifting subsequent items to make room |
| `L.pop()` | Remove and return the last item in the non-empty list `L` |
| `L.remove(v)` | Remove the first occurrence of `v` from the list `L` |
| `L.reverse()` | Reverse the order of items in the list `L` |
| `L.sort()` | Sort the items in `L` in ascending order |
| `L.sort(reverse=True)` | Sort the items in `L` in descending order |

# Where Did My List Go?

- Python programmers sometimes forget that many list methods return **`None`**
- As a result, lists sometimes seem to disappear:

```
>>> colors = 'red orange yellow green blue purple'.split()
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'purple']
>>> sorted_colors = colors.sort()
>>> print(sorted_colors)
None
```

- The string method `sort` changed the list but doesn't return the modified list:

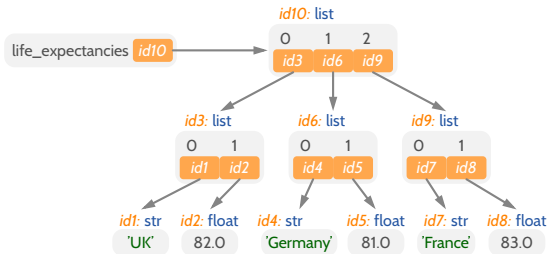
```
>>> colors
['blue', 'green', 'orange', 'purple', 'red', 'yellow']
```

**Why is it OK for *methods* (as opposed to *functions*) to do this?**

# Working with a List of Lists

- List items can have any type, in particular they can be lists themselves
- This is called a *nested* list
- For example, life expectancies in different countries

```
>>> life_expectancies = [['UK', 82.0], ['Germany', 81.0], ['France', 83.0]]
```

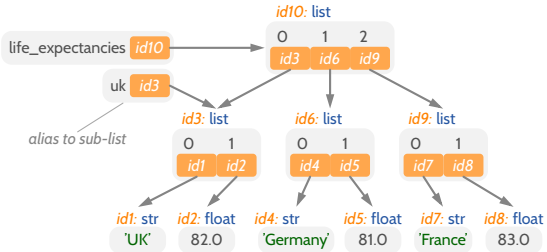



**Nesting is a common way to build data structures.**

# Referring to a Sub-list

- We can assign sub-lists to variables

```
>>> life_expectancies = [['UK', 82.0], ['Germany', 81.0], ['France', 83.0]]
>>> uk = life_expectancies[0]
>>> uk
['UK', 82.0]
>>> uk[0]
'UK'
>>> uk[1]
82.0
```




**Modifying the alias (reference) will change the original.**

# Exercises Lecture 8