Durham University

Intermediate Python

Dmitry Nikolaenko
Advanced Research Computing

September 29, 2022

Welcome and icebreaker

- Thank you for joining!
- 2-hour course (10am-12pm); break for 10 minutes around 11am
- Mute when not talking, but please do post in the chat or speak up when questions arise

- Introductions (post in the chat)

- Link to the exercises for today: https://colab.research.google.com/github/DurhamARC/Intermediate-Python/blob/main/exercises/intermediate_python_exercises.ipynb

Course structure

- 'Beginners Python' refresher
  - Loops, lists, functions
- 'Pythonic' concepts
  - List comprehension, ternary expressions, *args and **kwargs
  - Lambdas
- More advanced string manipulation
- Introduction to modules and packages
- Data structures and containers
  - Mutability
- Brief introduction to classes

Recap (a): Control Flow

- Conditional statements: if, elif, else
- Loop statements: for, while
- break, continue statements
  - How would the example's behaviour differ if break was swapped for continue?

```
for val in range(10):
    if val > 5:
        break
    else:
        print(val, end=' ')

# out: 0 1 2 3 4 5
```

Recap (a) cont.

- Nested loops

```
for x_axis in range(2):
    for y_axis in range(2):
        print('(' + str(x_axis) + ',' + str(y_axis) + ')')

# out:  (0,0)
#       (0,1)
#       (1,0)
#       (1,1)
```

But what is the *for* loop doing under the hood?

Control Flow (iterators)

1. iter() is called on the container object
2. This returns an iterator object
3. The iterator object defines a __next__() function
   - Facilitates access of elements one at a time
4. __next__() tells for loop when there are no more elements (raises StopIteration exception)

```
>>> uni = 'Durham'
>>> it = iter(uni)
>>> it
<str_iterator object at 0x10490ce20>
>>> next(it)
'D'
>>> next(it)
'u'
>>> next(it)
'r'
>>> next(it)
'h'
>>> next(it)
'a'
>>> next(it)
'm'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Recap (b): Lists

```
L = [1, 2, 3, 4, 5]
print(type(L))
# out:  <class 'list'>
```

- Quick and easy way to store objects
- Can contain objects of any type, or **even a mix of types**
  - Python's dynamic type system makes things easy!

```
random_stuff = [1, 'APPLES', 3.14, ['Mars', 'Venus', 'Pluto']]
print(random_stuff[3][2])
# OUT:  'Pluto'
```

Recap (b) cont.

- Easy to process lists using *for* loops
- Appending to lists

```
L = [1, 2, 3, 4, 5]
for val in L:
    print(val ** 2, end=" ")
# out: 1, 4, 9, 16, 25
```

```
chem_elements = ["oxygen"]
for i in range(2):
    chem_elements.append("hydrogen")
print(chem_elements)
# out: ['oxygen', 'hydrogen', 'hydrogen']
```

Recap (b) cont.: Semantics of slicing

- Simple to grab object in list if you know where it sits:

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
print(L[0])  # 1st letter
print(L[4])  # 5th letter
```

- Use ***:*** to grab a range defined subsection of the list (slicing):

```
start = 3
stop = 7
print(L[start:stop])  # items start to stop-1
# out:  ['d', 'e', 'f', 'g']
print(L[start:])      # items start to the end of list
# out:  ['d', 'e', 'f', 'g', 'h', 'i', 'j']
print(L[:stop])       # items from beginning of list to stop-1
# out:  ['a', 'b', 'c', 'd', 'e', 'f', 'g']
print(L[:])           # whole list
# out: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

Key - the ***:stop*** value represents the first value not in the slice

Recap (c): Functions

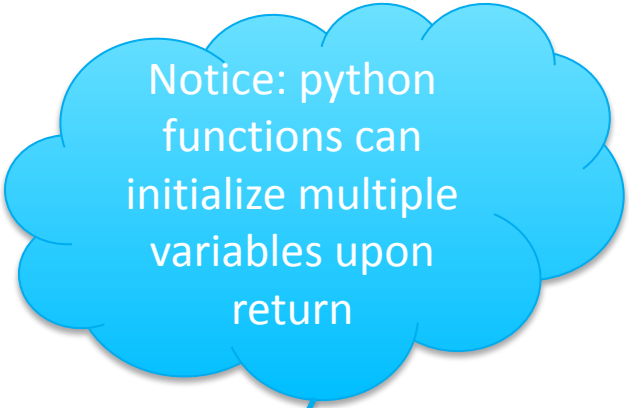
- Principle of **encapsulation**
  - Enable maintainability and readability alongside complexity
- **Reusable** code

- The def statement
- The importance of indentation
- The return statement

```
def increment(number):
    return number + 1

print("4+1 =", increment(4))
# out:  4+1 = 5
```

Building on the above (a)

- *args and **kwargs
  - *args for non-keyworded variables
  - *kwargs for keyworded variables




```
def print_shopping_list(*args):
    for arg in args:
        print(arg, end=" ")

print_shopping_list("spam", "eggs", "apples")

# out:  'spam eggs apples'
```

```
def print_shopping_list(**kwargs):
    for key, value in kwargs.items():
        print(key, "=", value, end=" ")

print_shopping_list(protein="eggs", fruit="apples")
# out:  'protein = eggs veg = apples'
```

Building on the above (b)

- Ternary expressions

```
hungry = True
state = "grumpy" if hungry else "content"
print(state)  # out:  "grumpy"
```

- List comprehension

```
multiples_of_three = [i for i in range(20) if i % 3 == 0]
print(multiples_of_three)
# out:  [0, 3, 6, 9, 12, 15, 18]
```

- Inbuilt functions that come with list
  - (We'll see more in the next slide)

```
L = [33, 84, 57, 11, 29, 0]
L.remove(57)
print(L)
# out:  [33, 84, 11, 29, 0]
```

More inbuilt list functions

- Inserting an element
- Reversing
  - Note: *reversed()* returns a 'reverse iterator' that then needs to be turned back into a list with *list()*
- Sorting
- Searching
- Emptying
- Removing duplicates

```
L = [33, 84, 11, 29, 0]

L.insert(2, 57)
print(L)
# out:  [33, 84, 57, 11, 29, 0]

print(list(reversed(L)))
# out:  [0, 29, 11, 57, 84, 33]

print(sorted(L))
# out:  [0, 11, 29, 33, 57, 84]

if 57 in L:
    position = L.index(57)
    print("57 is in the list. "
          "It is at position", position)
# out: 57 is in the list. It is at position 2

print(L.clear())  # out:  None

L2 = [1, 2, 2, 2, 3, 3]
print(list(set(L2)))  # out: [1, 2, 3]
```

Important distinction

- Some of these functions actively modified our list while others returned a copy of the modified list, leaving the original list untouched
  - So, to reverse our list permanently with *reversed()* we must use assignment
  - All *reversed()* does is return a reverse iterator!

```
L = [33, 84, 57, 11, 29, 0]
print(list(reversed(L)))
# out:  [0, 29, 11, 57, 84, 33]

print(L)
# out:  [33, 84, 57, 11, 29, 0]

L = list(reversed(L))

print(L)
# out:  [0, 29, 11, 57, 84, 33]
```

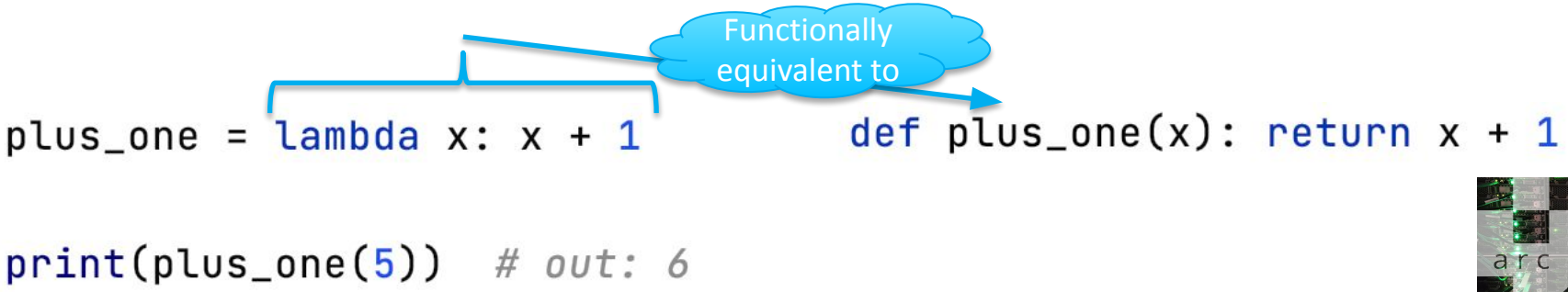
Better still…

- To avoid the explicit copy and assignment operations (potentially expensive) use *reverse()* instead of *reversed()*
  - Unlike *reversed()*, *reverse()* is a member function of the list itself
  - No single way is right: coding is about making good choices

```
L = [33, 84, 57, 11, 29, 0]
L.reverse()
print(L)
# out: [0, 29, 11, 57, 84, 33]
```

Building on the above (c): lambda functions

- Lambdas (sometimes anonymous functions) are one-line functions.
- Useful when you **don't want to use a function twice**
- Blueprint: lambda *arguments* : *expression*



- Lambdas can be used to **pass around functionality** as we will see next…

Lambda functions (continued)

- Map applies a function to all the items in a *list_of_inputs*.
  - **map(function_to_apply, list_of_inputs)**

- Where previously we would write:

```
items = [1, 2, 3, 4, 5]
squared = []

for i in items:
    squared.append(i ** 2)
print(squared)  # out: [1, 4, 9, 16, 25]
```

- Now this can be simplified to:

```
squared = list(map(lambda x: x ** 2, items))
print(squared)  # out: [1, 4, 9, 16, 25]
```

Lambda functions (continued)

- **Filter** creates a list of elements for which a function returns true:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)
# out:  [-5, -4, -3, -2, -1]
```

  - Equiv. non-lambda expression:

```
def is_less_than_zero(x):
    return True if x < 0 else False

negative_nums = []
for num in number_list:
    if is_less_than_zero(num):
        negative_nums.append(num)
print(negative_nums)  # out: [-5, -4, -3, -2, -1]
```

Mastering strings (a)

- Adjusting case
- Formatting strings

- None of these functions modify the original string. Instead, they return a **copy**. Modification requires assignment.

```
arc_update = "ThE HAmILton suPercompUTER is beiNg UPGraded"
print(arc_update.upper())
# out: THE HAMILTON SUPERCOMPUTER IS BEING UPGRADED
print(arc_update.title())
# out: The Hamilton Supercomputer Is Being Upgraded
print(arc_update.capitalize())
# out: The hamilton supercomputer is being upgraded

print(arc_update)
# out: ThE HAmILton suPercompUTER is beiNg UPGraded
```

```
arc_update = "    RSEs    "
print(arc_update.strip())
# out: 'RSEs'
print(arc_update.rstrip())
# out: '    RSEs'
print(arc_update.lstrip())
# out: 'RSEs    '
```

Durham University

Mastering strings (b)

- *find()*: return index of a substring
  - Returns -1 if substring not found
- Querying the existence of
- Replacing, splitting

```
line = 'the quick brown fox jumped over a lazy dog'
print(line.find('fox'))
# out: 16
print(line.find("nonexistent string"))
# out: -1

print(line.startswith("the"))
# out: true
print(line.endswith("fox"))
# out: false

print(line.replace("brown", "red"))
# out: the quick red fox jumped over a lazy dog

print(line.split())
# out: ['the', 'quick', 'brown', 'fox', 'jumped',
#       'over', 'a', 'lazy', 'dog']
```

Mastering strings (c)

- Index() is similar to find()
  - Returns index of substring
  - However, unlike find(), index() raises a ValueError **exception** when substring not found
  - Used alongside **exception handling**

```
line = 'the quick brown fox jumped over a lazy dog'

try:
    index = line.index('bear')
    print(index)
except ValueError:
    print("A bear is not mentioned in text")

# out:  A bear is not mentioned in text
```

An aside:

- The canonical way to search a string (if not interested in the index) is very simple:

```
line = 'the quick brown fox jumped over a lazy dog'

if "fox" in line:
    print("A fox has been seen")

# out:  A fox has been seen
```

Mastering strings (d)

- F-strings
  - F-strings provide a way to embed expressions inside string literals, using a minimal syntax
  - The expressions are evaluated at runtime and replaced with their values

```
interests = ["football", "zoom"]

print(f"Bob enjoys {interests[0]} and {interests[1]}")
# out:  Bob enjoys football and zoom


weekdays = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
for weekday in weekdays:
    print(f"Today is {weekday}")
# out: Today is Mon
#      Today is Tue
#      Today is Wed
#      ...


age = 70
print(f"Soon I'll be {age+1}!")
# out:  Soon I'll be 71
```

Exercises

Ask us questions!
We're very happy to setup breakout rooms upon request

Durham University

Coffee break

Durham University

Introduction to modules and packages

Durham University

Modules and packages

Namespaces

- Loading modules: the ***import*** statement
  - Explicit module import
  - Explicit module import by alias
  - Explicit import of module content

```
import math
math.cos(math.pi)
```

```
import numpy as np
np.cos(np.pi)
```

```
from math import cos, pi
cos(pi)
```

Other useful modules in the standard library

- Tools for interfacing with the operating system: **os**

Facilitates portability

```
import os
root = "/Users"
print(os.path.join(root, os.environ["USER"], "holiday_planning"))
# out  /Users/kgkc25/holiday_planning

os.listdir("Desktop")

if not os.path.exists("blahblahblah.txt"):
    print("File not found")
    exit(1)
```

Very useful when processing multiple data files

Using the *csv* module (part 1)

- Very convenient module for parsing and writing csv files

- Writing a csv

```
import csv

with open("example.csv", "w") as out_f:
    writer = csv.writer(out_f, delimiter=",")
    writer.writerow(["x_axis", "y_axis"])
    x_axis = [x * 0.1 for x in range(0, 100)]
    for x in x_axis:
        writer.writerow([x, math.cos(x)])
```

For the sake of visualization, here is the first part of the csv we just made:

x_axis	y_axis
0	1
0.1	0.99500417
0.2	0.98006658
0.3	0.95533649
0.4	0.92106099
0.5	0.87758256
0.6	0.82533561
0.7	0.76484219
0.8	0.69670671
0.9	0.62160997
1	0.54030231
1.1	0.45359612
1.2	0.36235775
1.3	0.26749883
1.4	0.16996714
1.5	0.0707372

Using the *csv* module (part 2)

- Now let's extract the value for y_axis when x_axis is 1.0 for the csv we just wrote:

```
import csv

with open("example.csv", "r") as in_file:
    reader = csv.reader(in_file, delimiter=",")
    next(reader)  # skip header
    for row in reader:
        if row[0] == "1.0":
            print(row[1])
            break

# out:  0.5403023058681398
```

Third-Party Modules

- Especially for data science coding, third party modules have a lot to offer
- Can be imported just as the built-in modules, but first the modules must be installed on your system.

- Python comes with a program called **pip** which will automatically fetch packages released and listed on PyPI
  - Example: pip install <*some-module*>

- If you don't have root permissions use **--user** option

```
[kqkc25@hamilton2 ~]$ module load python/3.6.8
[kqkc25@hamilton2 ~]$ pip3 install --user GitPython
```

Data structures

Durham University

Data structures: dictionaries

- Dictionaries are flexible mappings of keys to values
  - can be created via a comma-separated list of key:value pairs within curly braces:

```
numbers = {'one': 1, 'two': 2, 'three': 3}
```

  - Items are accessed and set via the indexing syntax used for lists and tuples, except here the index is not a zero-based order but valid key in the dictionary:

```
print(numbers['two'])  # out:  2
```

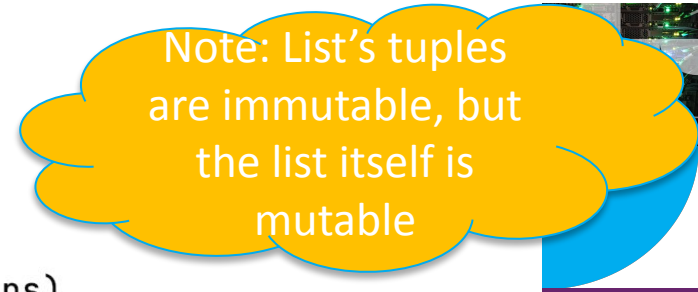
  - New items can be added to the dictionary using indexing:

```
numbers['ninety'] = 90
print(numbers)
# out:  {'one': 1, 'two': 2, 'three': 3, 'ninety': 90}
```

Data structures cont.: immutability

- All the data structures we have looked at are **mutable**
- Tuples exemplify **immutability**
- Typically, we have:
  - Lists for homogeneous data sequences (e.g., numbers, ingredients, names)
  - But tuples are ideal for heterogeneous data structures (where entries have different meanings - for example, coordinates)

```
location = (13, 88)
x_coordinate = location[0]
y_coordinate = location[1]

print(x_coordinate)  # out: 13
print(y_coordinate)

# location[0] = 4  # ERROR!

locations = [
    location,
    (14, 86),
    (15, 80)
]
print(locations)
# out: [(13, 88), (14, 86), (15, 80)]
```



Data structures cont.

- Namedtuple is handy, but also **immutable**
- Namedtuple is a factory function for making a tuple class
  - In the example, *NINumber* becomes a factory function that can encapsulate data from any employee

```
from collections import namedtuple
NINumber = namedtuple('national_insurance_number', 'name NI')
employee_data = NINumber('Simon', '12345678')
print(f'Employee {employee_data.name} has NI: {employee_data.NI}')
```

```
# assignment creates error!
# employee_data.NI = '4444'
```

This forces us to think about a core programming concept: classes…

Very brief introduction to classes

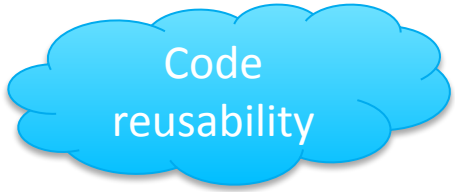
- Class = code template (like previously seen factory function)

```
class FootballTeam:
    players = ['Kane', 'Sterling', 'Pickford']

    def get_players(self):
        return self.players

england_team = FootballTeam()
print(england_team.get_players())
# out: ['Kane', 'Sterling', 'Pickford']
```

Ok… but how is this more interesting than a list?

Very brief introduction to classes

- We can **generalise** the template




```
class FootballTeam:
    def __init__(self, players):
        self.players = players

    def get_players(self):
        return self.players

england_team = FootballTeam(["Kane", "Sterling", "Pickford"])
print(england_team.get_players())
# out: ['Kane', 'Sterling', 'Pickford']
spanish_team = FootballTeam(["Moreno", "Llorente"])
print(spanish_team.get_players())
# out: ['Moreno', 'Llorente']
```

Fine… but doesn't this lack flexibility?

Very brief introduction to classes

- We can **encapsulate** complexity

```
class FootballTeam:
    def __init__(self, players):
        self.players = players

    def make_substitution(self, player_off, player_on):
        self.players = [player if player != player_off else player_on
                        for player in self.players]

    def get_players(self):
        return self.players

england_team = FootballTeam(["Kane", "Sterling", "Pickford"])
print(england_team.get_players())
# out: ['Kane', 'Sterling', 'Pickford']
england_team.make_substitution("Kane", "Grealish")
print(england_team.get_players())
# out: ['Grealish', 'Sterling', 'Pickford']
```

Exercises

Ask us questions!
We're very happy to setup breakout rooms upon request

Durham University

Durham University

Thank you!

- Feedback would really be appreciated: **https://bit.ly/arc_trainingfeedback**
- Other training courses at ARC
- RSE support

Solutions to the exercises can be found here:

https://colab.research.google.com/github/DurhamARC/Intermediate-Python/blob/main/exercises/intermediate_python_exercises_solutions.ipynb

Data structures cont.: mutable alternatives

- Same API as namedtuple, but mutable

```python
from dataclasses import dataclass

@dataclass(unsafe_hash=True)
class WeatherSystem:
    """Class for keeping track of local weather systems"""
    day_of_week: str
    temperature: float
    wind_speed: int
    rain: bool = False  # default value must come at end
```

- Adding member functions to this dataclass is also possible

```python
w = WeatherSystem('Monday', 35.0, 1)
print(w.day_of_week)  # out:  Monday

w.day_of_week = 'Tuesday'
w.temperature = 13.5
print(w.temperature)  # out:  13.5
```