Introduction to Python Programming

Kyle Jamieson

Networked Computer Systems Programme
Department of Computer Science
University College London

Setup and startup

```
% bash
$ cd ~
$ cp ~jamieson/pyintro_distribution.tgz .
$ gunzip pyintro_distribution.tgz
$ tar xvpf pyintro_distribution.tar
$ cd pyintro/src
```

- To start the Python interpreter, use `python_wrapper`:

```
$ python-wrapper
Python 2.6.2 (r262:71600, Sep  2 2009, 18:21:20)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
Type "help", "copyright", "credits" or "license" for
  more information.
>>>
```

- To exit the interpreter, type Ctrl+D
- Can type any Python expression at the primary command prompt (>>>)

Running Python scripts

- Start a text editor (a simple editor gedit, or your favorite), edit and save `hello.py` in `tubelab/src`

Contents of file `hello.py`:

```
$ cd pyintro/src
$ gedit hello.py
```

```
print “Hello, world.”
```

- Run your first program!

```
$ ./python-wrapper hello.py
Hello, world.
$
```

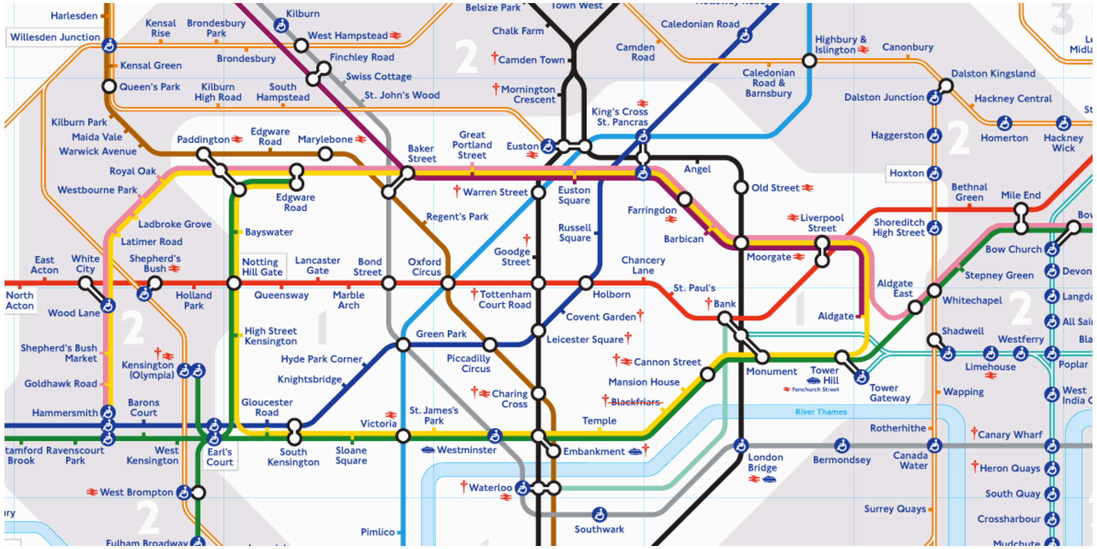
- Now, let’s start the Python interpreter and explore the language, starting with basic expressions and types

For further information

- Python tutorial
  - http://docs.python.org/tutorial
- Library reference
  - http://docs.python.org/library/index.html
- Language reference
  - http://docs.python.org/reference/index.html

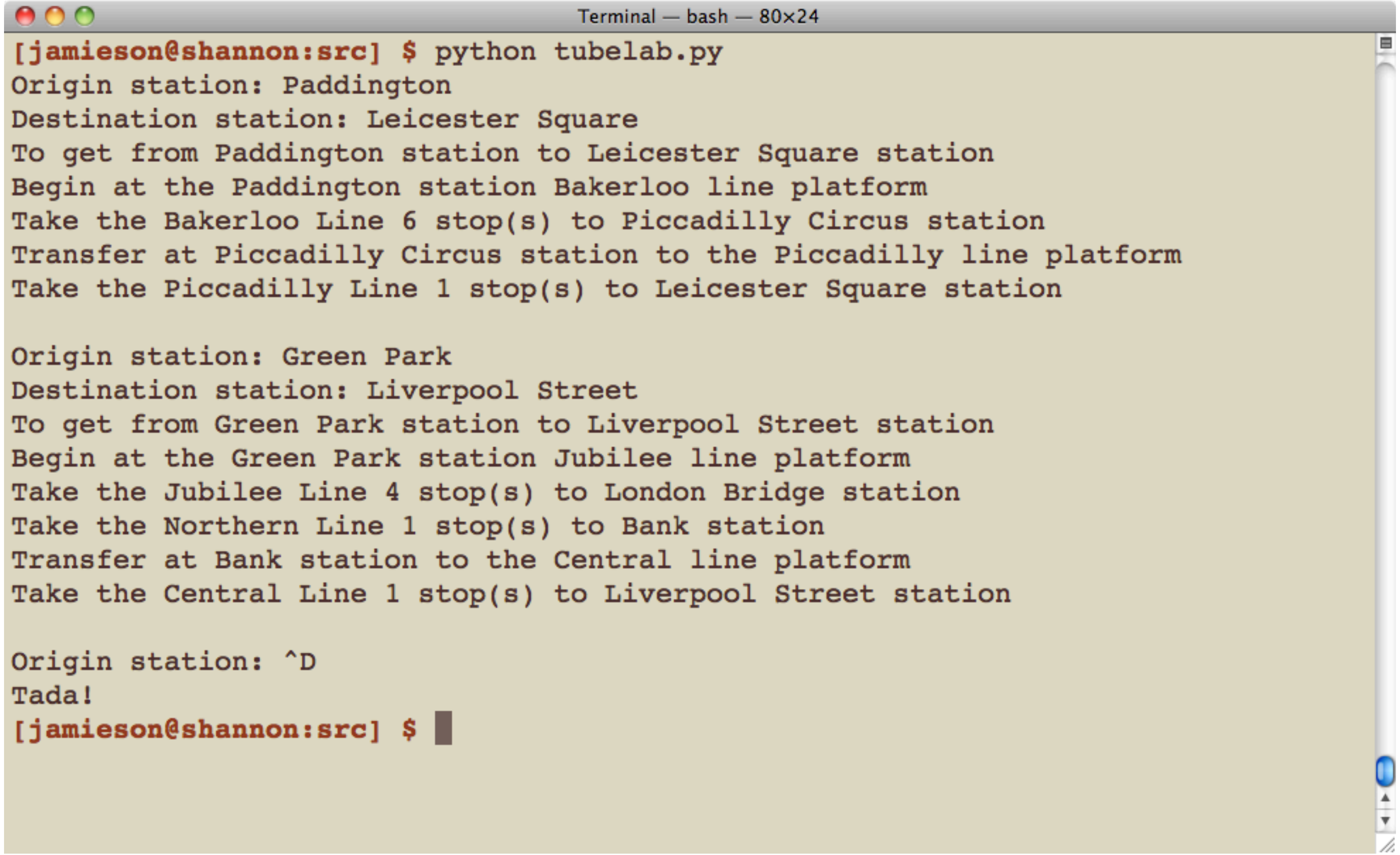
TubeLab

- **Objective:** Print out directions between pair of Zone 1 stations
  - Directions should correspond to shortest distance between the two stations, measured by sum of station stops and transfers between platforms
  - Represent this map with Python data structures
  - Breadth-first search to find shortest routes



http://www.tfl.gov.uk/assets/downloads/standard-tube-map.gif

A user’s interaction with TubeLab



```
[jamieson@shannon:src] $ python tubelab.py
Origin station: Paddington
Destination station: Leicester Square
To get from Paddington station to Leicester Square station
Begin at the Paddington station Bakerloo line platform
Take the Bakerloo Line 6 stop(s) to Piccadilly Circus station
Transfer at Piccadilly Circus station to the Piccadilly line platform
Take the Piccadilly Line 1 stop(s) to Leicester Square station

Origin station: Green Park
Destination station: Liverpool Street
To get from Green Park station to Liverpool Street station
Begin at the Green Park station Jubilee line platform
Take the Jubilee Line 4 stop(s) to London Bridge station
Take the Northern Line 1 stop(s) to Bank station
Transfer at Bank station to the Central line platform
Take the Central Line 1 stop(s) to Liverpool Street station

Origin station: ^D
Tada!
[jamieson@shannon:src] $ 
```

Two key data types

- *Station*: named on the tube map, contains platforms
  - Example: Euston Station has three platforms:
    - Two platforms on the **Northern** Line
    - One platform on the **Victoria** line

  - Example: Tottenham Court Road Station has one platform on the **Northern** line, one on the **Central** line

Tottenham
Court Road

  - Example: Great Portland Street Station has one platform on the **Hammersmith and City**, **Circle**, and **Metropolitan** lines

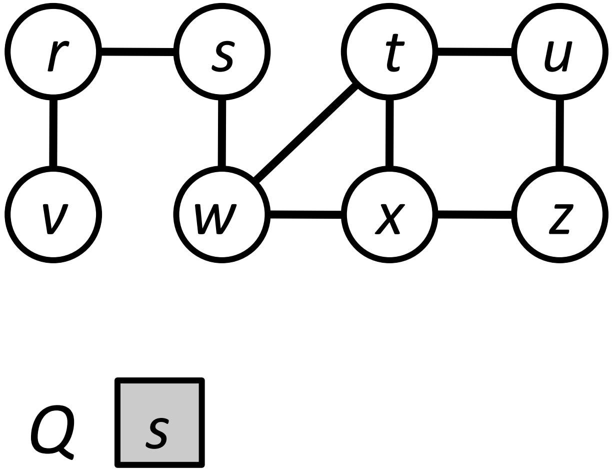
Great
Portland
Street

- *Platform*: associated with one or more lines
- *Line*: represent with Python strings

Breadth-first search (BFS)

- Explore graph, expanding frontier between undiscovered and discovered vertices uniformly across its breadth
- **Input:** Undirected graph $G = (V, E)$ and source vertex s
- **Output:** A breath-first tree with root s, containing shortest paths between s and any other vertex

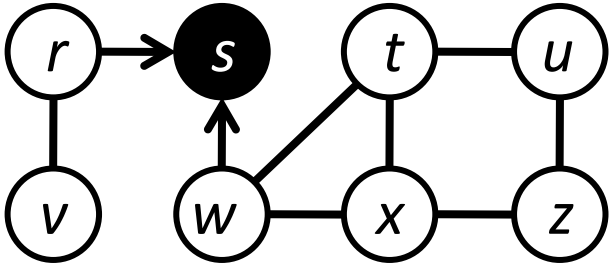
```
BFS($G$, $s$):
π($s$) ← None
$Q$ ← {$s$}
Visited ← {}
while $Q$ ≠ {}:
  $u$ ← $Q$.dequeue()
  Visited.add($u$)
  foreach $v \in$ Adj($u$):
    if $v \notin$ Visited and $v \notin Q$:
      $Q$.enqueue(v)
      π($v$) ← $u$
```

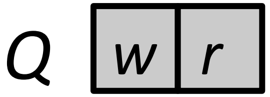


Breadth-first search (BFS)

- Explore graph, expanding frontier between undiscovered and discovered vertices uniformly across its breadth
- **Input:** Undirected graph $G = (V, E)$ and source vertex s
- **Output:** A breath-first tree with root s, containing shortest paths between s and any other vertex

```
BFS($G$, $s$):
π($s$) ← None
$Q$ ← {$s$}
Visited ← {s}
while $Q \neq$ {}:
  $u$ ← $Q$.dequeue()
  Visited.add($u$)
  foreach $v \in$ Adj($u$):
    if $v \notin$ Visited and $v \notin Q$:
      $Q$.enqueue(v)
      π($v$) ← $u$
```

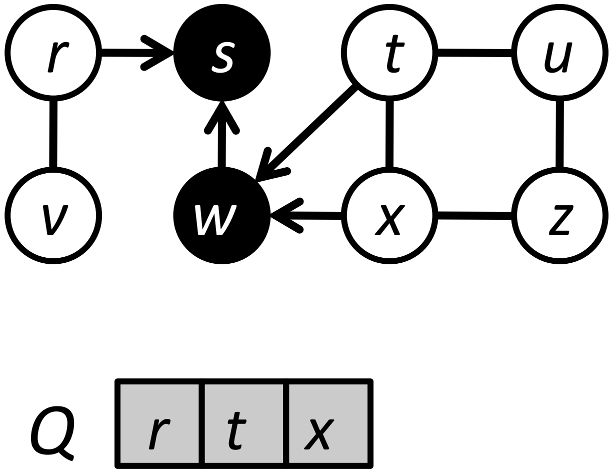




Breadth-first search (BFS)

- Explore graph, expanding frontier between undiscovered and discovered vertices uniformly across its breadth
- **Input:** Undirected graph *G* = (*V*, *E*) and source vertex *s*
- **Output:** A breath-first tree with root *s*, containing shortest paths between *s* and any other vertex

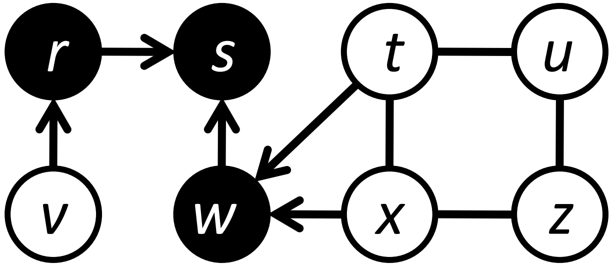
```
BFS($G$, $s$):
π($s$) ← None
$Q$ ← {$s$}
Visited ← {s}
while $Q \neq$ {}:
  $u$ ← $Q$.dequeue()
  Visited.add($u$)
  foreach $v \in$ Adj($u$):
    if $v \notin$ Visited and $v \notin Q$:
      $Q$.enqueue(v)
      π($v$) ← $u$
```



Breadth-first search (BFS)

- Explore graph, expanding frontier between undiscovered and discovered vertices uniformly across its <u>breadth</u>
- **Input:** Undirected graph $G = (V, E)$ and source vertex s
- **Output:** A breath-first tree with root s, containing shortest paths between s and any other vertex

```
BFS($G$, $s$):
π($s$) ← None
$Q$ ← {$s$}
Visited ← {s}
while $Q$ ≠ {}:
  $u$ ← $Q$.dequeue()
  Visited.add($u$)
  foreach $v \in$ Adj($u$):
    if $v \notin$ Visited and $v \notin Q$:
      $Q$.enqueue(v)
      π($v$) ← $u$
```

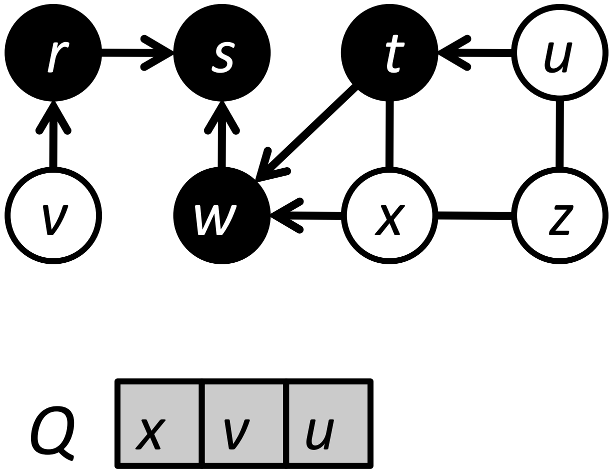


Q | *t* | *x* | *v* |

Breadth-first search (BFS)

- Explore graph, expanding frontier between undiscovered and discovered vertices uniformly across its breadth
- **Input:** Undirected graph $G = (V, E)$ and source vertex s
- **Output:** A breath-first tree with root s, containing shortest paths between s and any other vertex

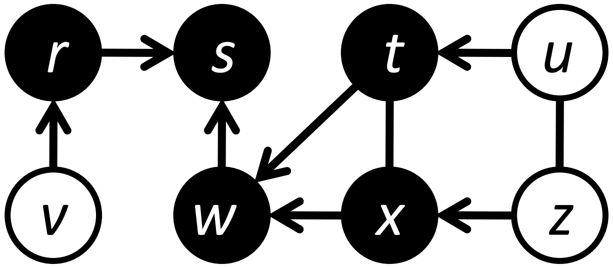
```
BFS(G, s):
π(s) ← None
Q ← {s}
Visited ← {s}
while Q ≠ {}:
  u ← Q.dequeue()
  Visited.add(u)
  foreach v ∈ Adj(u):
    if v ∉ Visited and v ∉ Q:
      Q.enqueue(v)
      π(v) ← u
```



Breadth-first search (BFS)

- Explore graph, expanding frontier between undiscovered and discovered vertices uniformly across its breadth
- **Input:** Undirected graph $G = (V, E)$ and source vertex s
- **Output:** A breath-first tree with root s, containing shortest paths between s and any other vertex

```
BFS(G, s):
π(s) ← None
Q ← {s}
Visited ← {s}
while Q ≠ {}:
  u ← Q.dequeue()
  Visited.add(u)
  foreach v ∈ Adj(u):
    if v ∉ Visited and v ∉ Q:
      Q.enqueue(v)
      π(v) ← u
```

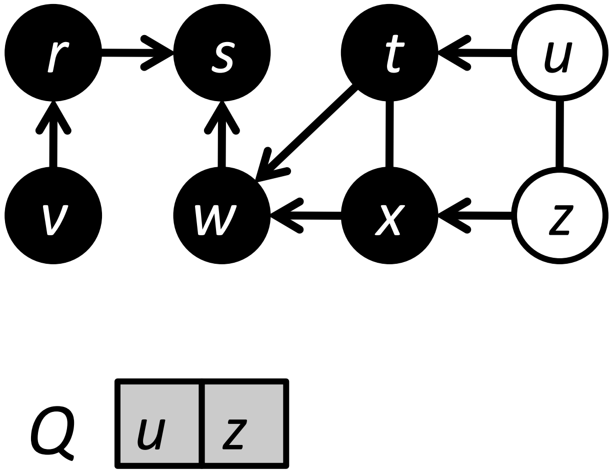


Q	v	u	z

Breadth-first search (BFS)

- Explore graph, expanding frontier between undiscovered and discovered vertices uniformly across its breadth
- **Input:** Undirected graph *G* = (*V*, *E*) and source vertex *s*
- **Output:** A breath-first tree with root *s*, containing shortest paths between *s* and any other vertex

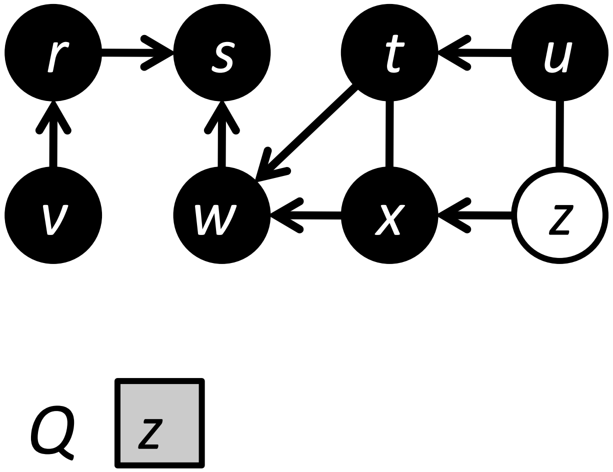
```
BFS($G$, $s$):
π($s$) ← None
$Q$ ← {$s$}
Visited ← {s}
while $Q \neq$ {}:
  $u$ ← $Q$.dequeue()
  Visited.add($u$)
  foreach $v \in$ Adj($u$):
    if $v \notin$ Visited and $v \notin Q$:
      $Q$.enqueue(v)
      π($v$) ← $u$
```



Breadth-first search (BFS)

- Explore graph, expanding frontier between undiscovered and discovered vertices uniformly across its breadth
- **Input:** Undirected graph $G = (V, E)$ and source vertex s
- **Output:** A breath-first tree with root s, containing shortest paths between s and any other vertex

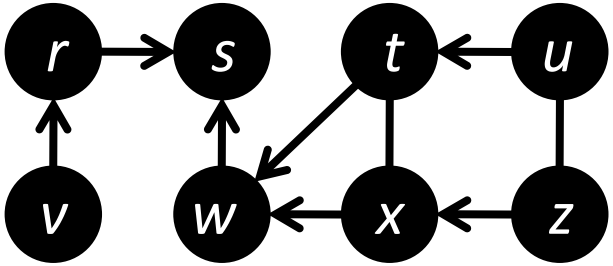
```
BFS(G, s):
π(s) ← None
Q ← {s}
Visited ← {s}
while Q ≠ {}:
  u ← Q.dequeue()
  Visited.add(u)
  foreach v ∈ Adj(u):
    if v ∉ Visited and v ∉ Q:
      Q.enqueue(v)
      π(v) ← u
```



Breadth-first search (BFS)

- Explore graph, expanding frontier between undiscovered and discovered vertices uniformly across its <u>breadth</u>
- **Input:** Undirected graph $G = (V, E)$ and source vertex s
- **Output:** A breath-first tree with root s, containing shortest paths between s and any other vertex

```
BFS($G$, $s$):
$\pi(s) \leftharpoonup$ None
$Q \leftharpoonup \{s\}$
Visited $\leftharpoonup$ {s}
while $Q \neq \{\}$:
  $u \leftharpoonup Q$.dequeue()
  Visited.add($u$)
  foreach $v \in$ Adj($u$):
    if $v \notin$ Visited and $v \notin Q$:
      $Q$.enqueue(v)
      $\pi(v) \leftharpoonup u$
```

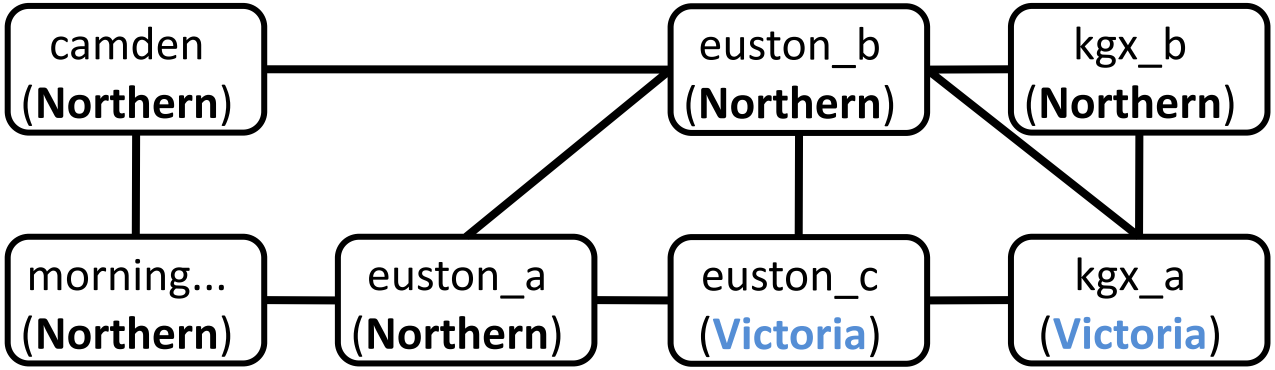


Q

From tube map to graph representation

- Each vertex in the abstract graph corresponds to a Platform
- Each edge is either a transfer or a trip between platforms of different stations on the same line
- Examples:
  - (camden, euston_b): Northern line south one stop
  - (euston_b, euston_c): Transfer at Euston Station to the Victoria line
- BFS happens in this graph, breadth-first tree constructed using `Platform.set_predecessor(Platform)`

Camden Town
Camden Road
Mornington Crescent
King's Cross St. Pancras
Euston



Code walkthru: Stations

- A Station groups a number of platforms together
- The iterator returned by platforms(self) yields all Platforms contained within the Station

`station.py`

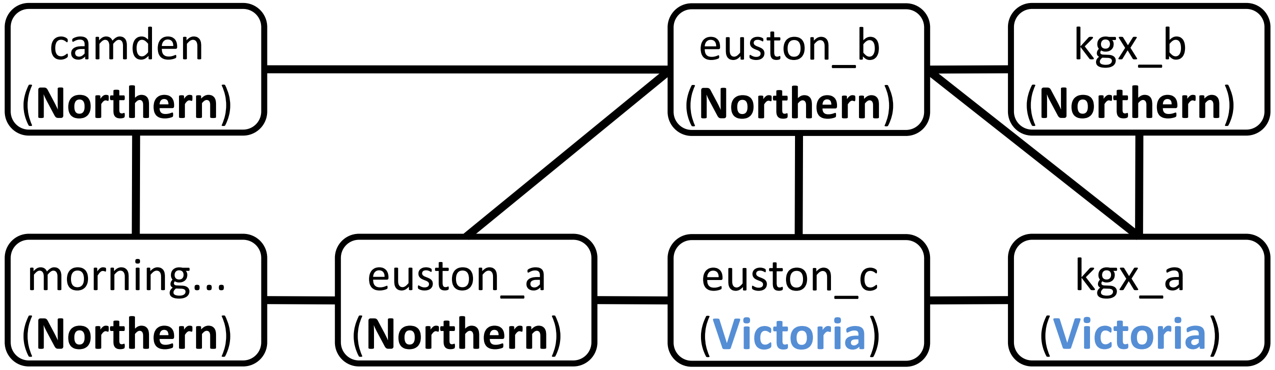
```
class Station
  def __init__(self, name, platforms) # list of platforms
  def __str__(self) # return a string representation
  def platforms(self) # returns a generator ==> platform
```

Code walkthru: Platforms

platform.py

```
class Platform
  def __init__(self, lines) # list of strings: which lines?
  def set_station(self, station) # set containing station
  def add_neighbor(self, line, other_platform)
  def set_predecessor(self, platform) # for BFS
  def set_predecessor_line(self, line) # for BFS
  def neighbors(self) # returns an iterator ==> (platform, line)
```

- Convention: the iterator returned by `neighbors(self)` yields `(Platform, None)` for transfers within a Station



Code walkthru: Putting it together

tubedata.py

```
from platform import *
from station import *

northern = 'Northern'
victoria = 'Victoria’
central = 'Central’

tott_ct_rd_a = Platform([northern])
tott_ct_rd_b = Platform([central])
tott_ct_rd_sta = Station('Tottenham
 Court Road’, [tott_ct_rd_a,
 tott_ct_rd_b])

goodge_st = Platform([northern])
goodge_st_sta = Station('Goodge
 Street', [goodge_st])
```

```
warren_st = Platform([northern,
 victoria])
warren_st_sta = Station('Warren
 Street', [warren_st])

euston_a = Platform([northern])
euston_b = Platform([northern])
euston_c = Platform([victoria])
euston_sta = Station('Euston',
 [euston_a, euston_b, euston_c])

connect(northern, [waterloo_b,
 embankment_a, charing_cross_a,
 leicester_sq, tott_ct_rd_a,
 goodge_st, warren_st, euston_a,
 mornington_crescent, camden_town])
```

Exercise: fire up Python, and print all neighbors of the Victoria line platform in Euston Station (euston_c):

```
$ cd tubelab
$ python-wrapper
>>> from tubedata import *
>>>
```

Exercise: fire up Python, and print all platforms at Euston Station (euston_sta):

```
$ cd tubelab
$ python-wrapper
>>> from tubedata import *
>>>
```

Code walkthru: Main program

- The program entry point, command line interface (CLI), and shortest-paths calculator are in file `tubelab.py`
  - `bfs_directions`: computes shortest-paths with BFS
  - `completer`: allows user to use tab to complete
  - Entry point: runs a while loop until Ctrl+D or Ctrl+C input from user, takes input, passes it to `bfs_directions`

> **Main lab assignment**
>
> Implement BFS's "inner loop" in `bfs_directions`.

- To run/test:

```
$ python-wrapper tubelab.py
```