Python Programming

Packing/Unpacking, `*args`/`**kwargs`, Lambdas and Uniform Cost Search

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

September 07, 2023

Packing and Unpacking: Tuples

- Following creates a tuple: `x = (10, 20, 30)`

Packing and Unpacking: Tuples

- Following creates a tuple: `x = (10, 20, 30)`
- What about the following: `y = 10, 20, 30`

Packing and Unpacking: Tuples

- Following creates a tuple: `x = (10, 20, 30)`
- What about the following: `y = 10, 20, 30`
  also creates a tuple; verify with `type(y)`
- Such comma separated list is *packed* into a tuple automatically and this tuple is then assigned to variable `y`

Packing and Unpacking: Tuples

- Following creates a tuple: `x = (10, 20, 30)`
- What about the following: `y = 10, 20, 30`
  also creates a tuple; verify with `type(y)`
- Such comma separated list is *packed* into a tuple automatically and this tuple is then assigned to variable `y`
- Now what about: `a, b, c = y`
  print the values of `a`, `b`, and `c`

Packing and Unpacking: Tuples

- Following creates a tuple: `x = (10, 20, 30)`
- What about the following: `y = 10, 20, 30`
  also creates a tuple; verify with `type(y)`
- Such comma separated list is *packed* into a tuple automatically and this tuple is then assigned to variable `y`
- Now what about: `a, b, c = y`
  print the values of `a`, `b`, and `c`
- The tuple `y` is automatically *unpacked*
  then the following happens: `a, b, c = 10, 20, 30`

Packing and Unpacking: Iterable

- Put an asterisk (*) in front of a variable to mark it as an iterable (e.g. list)
- We can assign (pack) multiple values into such variables

```
*a = 10, 20, 30 # print and check type of a
```

Packing and Unpacking: Iterable

- Put an asterisk (`*`) in front of a variable to mark it as an iterable (e.g. list)
- We can assign (pack) multiple values into such variables
  ```
  *a = 10, 20, 30 # print and check type of a
  ```
- What about: `a, b, *c = 10, 20, 30, 40, 50`

Packing and Unpacking: Iterable

- Put an asterisk (*) in front of a variable to mark it as an iterable (e.g. list)
- We can assign (pack) multiple values into such variables

```
*a = 10, 20, 30 # print and check type of a
```

- What about: `a, b, *c = 10, 20, [30, 40, 50]`

```
# print and test
```

Packing and Unpacking: Iterable

- Put an asterisk (`*`) in front of a variable to mark it as an iterable (e.g. list)
- We can assign (pack) multiple values into such variables

  ```
  *a = 10, 20, 30 # print and check type of a
  ```

- What about: `a, b, *c = 10, 20, [30, 40, 50]`

  ```
  # print and test
  ```

- What about: `a, *b, c = 10, 20, 30, 40, 50`

Packing and Unpacking: Iterable

- Put an asterisk (`*`) in front of a variable to mark it as an iterable (e.g. list)
- We can assign (pack) multiple values into such variables

  `*a = 10, 20, 30 # print and check type of a`

- What about: `a, b, *c = 10, 20, [30, 40, 50]`

  `# print and test`

- What about: `a, *b, c = 10, [20, 30, 40], 50`

  `# print and test`

Packing and Unpacking: Iterable

- Put an asterisk (`*`) in front of a variable to mark it as an iterable (e.g. list)
- We can assign (pack) multiple values into such variables

```
*a = 10, 20, 30 # print and check type of a
```

- What about: `a, b, *c = 10, 20, [30, 40, 50]` `# print and test`
- What about: `a, *b, c = 10, [20, 30, 40], 50` `# print and test`
- What about: `a, *b, c = 10, 20 # print and test`

Packing and Unpacking: Iterable

- Put an asterisk (*) in front of a variable to mark it as an iterable (e.g. list)
- We can assign (pack) multiple values into such variables

  `*a = 10, 20, 30 # print and check type of a`

- What about: `a, b, *c = 10, 20, [30, 40, 50]`

  `# print and test`

- What about: `a, *b, c = 10, [20, 30, 40], 50`

  `# print and test`

- What about: `a, *b, c = 10, 20 # print and test`

- We can unpack an iterable by placing an asterisk in front of it

```
x = [1, 2, 3]
print(x) # as a list
print(*x) # individual elements
# same as: print(x[0], x[1], x[2])
```

Packing and Unpacking: Functions

- The packing/unpacking is ubiquitous in Python
- Returning multiple values from a function

Packing and Unpacking: Functions

- The packing/unpacking is ubiquitous in Python
- Returning multiple values from a function

- It can be used to pass variable number of arguments

```
def add(*args):
    sum = 0
    for val in args:
        sum += val
    return sum
```

Packing and Unpacking: Functions

- The packing/unpacking is ubiquitous in Python
- Returning multiple values from a function

- It can be used to pass variable number of arguments

```
def add(*args):
    sum = 0
    for val in args:
        sum += val
    return sum
add(1, 2)
add(1, 2, 3 ,5)
add() # also possible
```

Packing and Unpacking: Functions

- The packing/unpacking is ubiquitous in Python
- Returning multiple values from a function

- It can be used to pass variable number of arguments

```
def add(*args):
    sum = 0
    for val in args:
        sum += val
    return sum
add(1, 2)
add(1, 2, 3 ,5)
add() # also possible
```

- The mandatory arguments are placed at the front

```
def myFunc(arg1, arg2, *args):
    print(f"called with {2+len(args)} arguments")
```

Keyworded Arguments

- *Positional Argument*: Classical way of passing arguments

```
def foo(x, y):
    print(f'value of x is {x} and y is {y}')
...
foo(10, 20) # x=10, y=20
```

Keyworded Arguments

- *Positional Argument*: Classical way of passing arguments

```
def foo(x, y):
    print(f'value of x is {x} and y is {y}')
...
foo(10, 20) # x=10, y=20
```

- Arguments are ordered[1]

[1] Things in `*args` are also ordered

Keyworded Arguments

- *Positional Argument*: Classical way of passing arguments

```
def foo(x, y):
    print(f'value of x is {x} and y is {y}')
...
foo(10, 20) # x=10, y=20
```

- Arguments are ordered[1]

- *Keyword Argument*: We can be explicit about the associations

```
foo(x=10, y=20)
```

[1] Things in `*args` are also ordered

Keyworded Arguments

- *Positional Argument*: Classical way of passing arguments

```python
def foo(x, y):
    print(f'value of x is {x} and y is {y}')
...
foo(10, 20) # x=10, y=20
```

- Arguments are ordered[1]

- *Keyword Argument*: We can be explicit about the associations

```python
foo(x=10, y=20)
```

- With explicit associations, we can pass arguments out of order

```python
foo(y=20, x=10)
```

[1] Things in `*args` are also ordered

Keyworded Arguments

- *Positional Argument*: Classical way of passing arguments

```
def foo(x, y):
    print(f'value of x is {x} and y is {y}')
...
foo(10, 20) # x=10, y=20
```

- Arguments are ordered[1]

- *Keyword Argument*: We can be explicit about the associations

```
foo(x=10, y=20)
```

- With explicit associations, we can pass arguments out of order

```
foo(y=20, x=10)
```

- We can also do the following:

```
vals = {'x': 10, 'y': 20} # keys are arguments (as str)
```

[1] Things in `*args` are also ordered

Keyworded Arguments

- *Positional Argument*: Classical way of passing arguments

```
def foo(x, y):
    print(f'value of x is {x} and y is {y}')
...
foo(10, 20) # x=10, y=20
```

- Arguments are ordered[1]

- *Keyword Argument*: We can be explicit about the associations

```
foo(x=10, y=20)
```

- With explicit associations, we can pass arguments out of order

```
foo(y=20, x=10)
```

- We can also do the following:

```
vals = {'x': 10, 'y': 20} # keys are arguments (as str)
foo(**vals) # unpacks the dict
```

[1] Things in `*args` are also ordered

More on Keyworded Arguments

- We can pass keyworded variable length of arguments to a function

```
def foo(**kwargs): # received as a dict object
    for key, value in kwargs.items():
        print(f'{key} = {value}')

foo(x=10, z=30, y=20)
```

More on Keyworded Arguments

- We can pass keyworded variable length of arguments to a function

```
def foo(**kwargs): # received as a dict object
    for key, value in kwargs.items():
        print(f'{key} = {value}')

foo(x=10, z=30, y=20)
```

- Careful with the ordering of `*args`, `**kwargs` and formal args

```
def foo(arg1, arg2, *args, **kwargs): # note the order
    ...
```

Creating Function Alias

- We can create aliases of a function, just like any variable

```python
def f(x):
    print(x)


h = f # h is now an alias of function f

# now both f and h can be called

h(10) # same as calling f(10)
```

Lambdas

- Anonymous function having only a single statement
- Typically used for temporary purposes
- Syntax: `lambda args: expression`

Lambdas

- Anonymous function having only a single statement
- Typically used for temporary purposes
- Syntax: `lambda args: expression`
- We may assign it to some alias

```
f = lambda a: print(a)

f(10)
```

Lambdas

- Anonymous function having only a single statement
- Typically used for temporary purposes
- Syntax: `lambda args: expression`
- We may assign it to some alias

```
f = lambda a: print(a)

f(10)
```

Think of this as

```
def some_name(a):
    print(a)
f = some_name
f(10)
```

Lambdas

- Anonymous function having only a single statement
- Typically used for temporary purposes
- Syntax: `lambda args: expression`
- We may assign it to some alias

```
f = lambda a: print(a)

f(10)
```

Think of this as

```
def some_name(a):
    print(a)
f = some_name
f(10)
```

- Another example:

```
f = lambda a,b: a+b

# evaluated expression is returned
r = f(10, 20)
print(r) # 30
```

Lambdas

- Anonymous function having only a single statement
- Typically used for temporary purposes
- Syntax: `lambda args: expression`
- We may assign it to some alias

```
f = lambda a: print(a)

f(10)
```

Think of this as

```
def some_name(a):
    print(a)
f = some_name
f(10)
```

- Another example:

```
f = lambda a,b: a+b

# evaluated expression is returned
r = f(10, 20)
print(r) # 30
```

Think of this as

```
def add(a, b):
    return a+b
f = add
r = f(10, 20)
print(r)
```

Using Lambdas: an Example

- Suppose we want to sort a list 2D points
  `P = [(1,2), (3,0), (2,2), (2,1)]`

Using Lambdas: an Example

- Suppose we want to sort a list 2D points
  `P = [(1,2), (3,0), (2,2), (2,1)]`
- We may use the built-in `sort()` method

```
P.sort()
print(P) # [(1, 2), (2, 1), (2, 2), (3, 0)]
```

Using Lambdas: an Example

- Suppose we want to sort a list 2D points
  `P = [(1,2), (3,0), (2,2), (2,1)]`
- We may use the built-in `sort()` method

```
P.sort()
print(P) # [(1, 2), (2, 1), (2, 2), (3, 0)]
```

- Sorts in lexicographic order starting with x-coordinate
- What if we want to sort by y-coordinates?

Using Lambdas: an Example

- Suppose we want to sort a list 2D points
  `P = [(1,2), (3,0), (2,2), (2,1)]`
- We may use the built-in `sort()` method

```
P.sort()
print(P) # [(1, 2), (2, 1), (2, 2), (3, 0)]
```

- Sorts in lexicographic order starting with x-coordinate
- What if we want to sort by y-coordinates?
- The built-in `sort()` method can accept an argument which specifies the comparison *key*: `sort(key=some_mapping_func)`

Using Lambdas: an Example

- Suppose we want to sort a list 2D points
  `P = [(1,2), (3,0), (2,2), (2,1)]`
- We may use the built-in `sort()` method

```
P.sort()
print(P) # [(1, 2), (2, 1), (2, 2), (3, 0)]
```

- Sorts in lexicographic order starting with x-coordinate
- What if we want to sort by y-coordinates?
- The built-in `sort()` method can accept an argument which specifies the comparison *key*: `sort(key=some_mapping_func)`
- Given an element, the mapping function returns a value that is actually used in the sorting comparison
- For each point `a`, `a[1]` is its y-coordinate value

Using Lambdas: an Example

- Suppose we want to sort a list 2D points
  `P = [(1,2), (3,0), (2,2), (2,1)]`
- We may use the built-in `sort()` method

```
P.sort()
print(P) # [(1, 2), (2, 1), (2, 2), (3, 0)]
```

- Sorts in lexicographic order starting with x-coordinate
- What if we want to sort by y-coordinates?
- The built-in `sort()` method can accept an argument which specifies the comparison *key*: `sort(key=some_mapping_func)`
- Given an element, the mapping function returns a value that is actually used in the sorting comparison
- For each point `a`, `a[1]` is its y-coordinate value

```
P.sort(key=lambda a: a[1]) # only by y-coordinates
print(P) # [(3, 0), (2, 1), (1, 2), (2, 2)]
```

Using Lambdas: More Examples

```
P = [(1,2), (3,0), (2,2), (2,1)]

min(P, key=lambda a: a[1])  # point having min y
max(P, key=lambda a: a[0]+a[1])  # point having max (x+y)
```

Using Lambdas: More Examples

```
P = [(1,2), (3,0), (2,2), (2,1)]

min(P, key=lambda a: a[1])  # point having min y
max(P, key=lambda a: a[0]+a[1])  # point having max (x+y)

students = []
students.append( {'name': 'abcd', 'marks': 90} )
students.append( {'name': 'wxyz', 'marks': 40} )
students.append( {'name': 'mnop', 'marks': 70} )
```

Using Lambdas: More Examples

```
P = [(1,2), (3,0), (2,2), (2,1)]

min(P, key=lambda a: a[1])  # point having min y
max(P, key=lambda a: a[0]+a[1])  # point having max (x+y)


students = []
students.append( {'name': 'abcd', 'marks': 90} )
students.append( {'name': 'wxyz', 'marks': 40} )
students.append( {'name': 'mnop', 'marks': 70} )

# student having min marks
min_student = min(students, key=lambda s: s['marks'])
print(min_student)
```

Using Lambdas: More Examples

```
P = [(1,2), (3,0), (2,2), (2,1)]

min(P, key=lambda a: a[1])  # point having min y
max(P, key=lambda a: a[0]+a[1])  # point having max (x+y)


students = []
students.append( {'name': 'abcd', 'marks': 90} )
students.append( {'name': 'wxyz', 'marks': 40} )
students.append( {'name': 'mnop', 'marks': 70} )

# student having min marks
min_student = min(students, key=lambda s: s['marks'])
print(min_student)

# sort students by names
students.sort(key=lambda s: s['name'])
print(students)
```

Weighted Graphs

Our previous graph class

```
class Graph:
    def __init__(self, n):
        self._vertex_count = n
        self._adj_list = [ [] for _ in range(n) ]

    def add_edge(self, u, v):
        self._adj_list[u].append( v )
        self._adj_list[v].append( u )

    def get_neighbours(self, v):
        return self._adj_list[v]
```

Weighted Graphs

Store weights for each edge

```
class Graph:
    def __init__(self, n):
        self._vertex_count = n
        self._adj_list = [ [] for _ in range(n) ]

    def add_edge(self, u, v, weight): # new parameter
        self._adj_list[u].append( (v, weight) ) # tuple
        self._adj_list[v].append( (u, weight) )

    def get_neighbours(self, v):
        return self._adj_list[v] # returns a list of tuples
```

Weighted Graphs

Store weights for each edge

```
class Graph:
    def __init__(self, n):
        self._vertex_count = n
        self._adj_list = [ [] for _ in range(n) ]

    def add_edge(self, u, v, weight=1): # default value
        self._adj_list[u].append( (v, weight) ) # tuple
        self._adj_list[v].append( (u, weight) )

    def get_neighbours(self, v):
        return self._adj_list[v] # returns a list of tuples
```

Weighted Graphs

Store weights for each edge: using dictionary

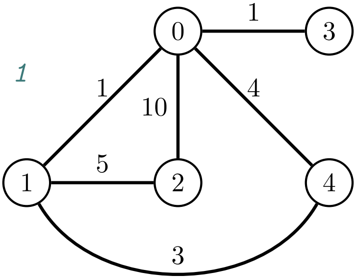
```
class Graph:
    def __init__(self, n):
        self._vertex_count = n
        self._adj_list = [ [] for _ in range(n) ]

    def add_edge(self, u, v, weight=1): # default value
        self._adj_list[u].append({'node': v,'weight': weight})
        self._adj_list[v].append({'node': u,'weight': weight})

    def get_neighbours(self, v):
        return self._adj_list[v] # returns a list of tuples
```

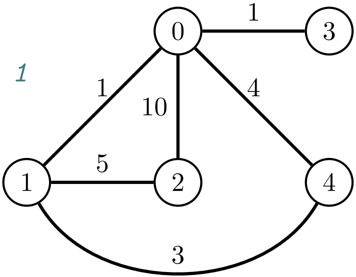
Using the Modified Graph Class

```
g = Graph(5)
g.add_edge(0, 1) # default weight 1
g.add_edge(0, 2, 10)
g.add_edge(0, 3, 1)
g.add_edge(0, 4, 4)
g.add_edge(1, 2, 5)
g.add_edge(1, 4, 3)
```



Using the Modified Graph Class

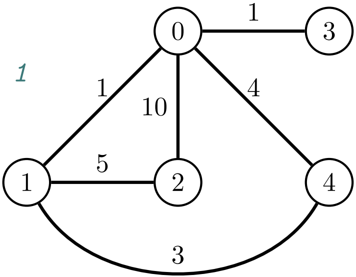
```
g = Graph(5)
g.add_edge(0, 1) # default weight 1
g.add_edge(0, 2, 10)
g.add_edge(0, 3, 1)
g.add_edge(0, 4, 4)
g.add_edge(1, 2, 5)
g.add_edge(1, 4, 3)
```



How to get the neighbour of node 1 having minimum edge weight?

Using the Modified Graph Class

```
g = Graph(5)
g.add_edge(0, 1) # default weight 1
g.add_edge(0, 2, 10)
g.add_edge(0, 3, 1)
g.add_edge(0, 4, 4)
g.add_edge(1, 2, 5)
g.add_edge(1, 4, 3)
```

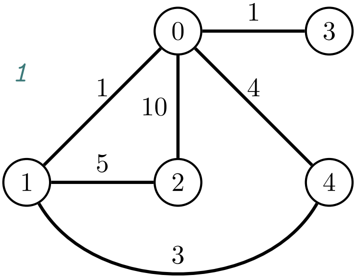


How to get the neighbour of node 1 having minimum edge weight?

```
neighbours = g.get_neighbours(1) # list of dicts
```

Using the Modified Graph Class

```
g = Graph(5)
g.add_edge(0, 1) # default weight 1
g.add_edge(0, 2, 10)
g.add_edge(0, 3, 1)
g.add_edge(0, 4, 4)
g.add_edge(1, 2, 5)
g.add_edge(1, 4, 3)
```

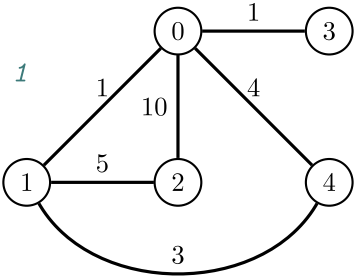


How to get the neighbour of node 1 having minimum edge weight?

```
neighbours = g.get_neighbours(1) # list of dicts
min_neighbour = min(neighbours,key=lambda x: x['weight'])
```

Using the Modified Graph Class

```
g = Graph(5)
g.add_edge(0, 1) # default weight 1
g.add_edge(0, 2, 10)
g.add_edge(0, 3, 1)
g.add_edge(0, 4, 4)
g.add_edge(1, 2, 5)
g.add_edge(1, 4, 3)
```



How to get the neighbour of node 1 having minimum edge weight?

```
neighbours = g.get_neighbours(1) # list of dicts
min_neighbour = min(neighbours,key=lambda x: x['weight'])
print( min_neighbour['node'] )
```

Uniform Cost Search

Complete code: graph_UCS.py