MONASH College

MCD4710

Introduction to algorithms and programming

Lecture 17

Advanced Python

Overview

- Utilise conventions around Boolean expressions
- Write simple list comprehensions
- Demonstrate utility of nested functions

Disclaimer

Techniques shown here

- …can make your life simpler
- …are used in some of the programs presented in the remainder of the unit

But (outside of this week), we won't specifically ask you to use them

If you still struggle with basic programming techniques, work on those first

Boolean values and expressions

Let's start with a quiz

MONASH College

```
def insert(i, lst):
    n = len(lst)
    j = i
    while lst[j + 1] < lst[j] and j < n-1:
        lst[j + 1], lst[j] = lst[j], lst[j+1]
        j += 1
```

What is the state of lst after executing these statements?

```
>>> lst = [34, 27, 1, 2, 3]
>>> insert(1, lst)
```

a) [27, 1, 2, 3, 34]

b) [34, 1, 2, 3, 27]

c) [1, 2, 3, 27, 34]

d) Index Error

e) None of the above

1. Visit https://flux.qa
2. Log in using your Authcate details (not required if you're already logged in to Monash)
3. Touch the + symbol and enter the code: HHYBXU
4. Answer questions when they pop up.

Evaluation of 1st part of the loop condition is invalid for the last index

```
def insert(i, lst):
    n = len(lst)
    j = i
    while lst[j + 1] < lst[j] and j < n-1:
        lst[j + 1], lst[j] = lst[j], lst[j+1]
        j += 1
```

```
>>> lst = [34, 27, 1, 2, 3]
>>> insert(1, lst)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lecture17.py", line 5, in insert
    while lst[j + 1] < lst[j] and j < n-1:
IndexError: list index out of range
```

Can move the order condition inside the loop body for conditional break

```
def insert(i, lst):
    n = len(lst)
    j = i
    while j < n-1:
        if lst[j + 1] >= lst[j]:
            break
        lst[j + 1], lst[j] = lst[j], lst[j+1]
        j += 1
```

```
>>> lst = [34, 27, 1, 2, 3]
>>> insert(1, lst)
>>> lst
[34, 1, 2, 3, 27]
```

That works, but is hard to understand…

Rule of thumb: the more "nesting", the harder to read

Short-circuiting allows easy to read conditional evaluation

…if one is used to this behaviour of logical operators

```
def insert(i, lst):
    n = len(lst)
    j = i
    while j < n-1 and lst[j + 1] < lst[j]:
        lst[j + 1], lst[j] = lst[j], lst[j+1]
        j += 1
```

second and-operand only evaluated if first is true

```
>>> lst = [34, 27, 1, 2, 3]
>>> insert(1, lst)
>>> lst
[34, 1, 2, 3, 27]
```

Short-circuiting allows easy to read conditional evaluation

…if one is used to this behaviour of logical operators

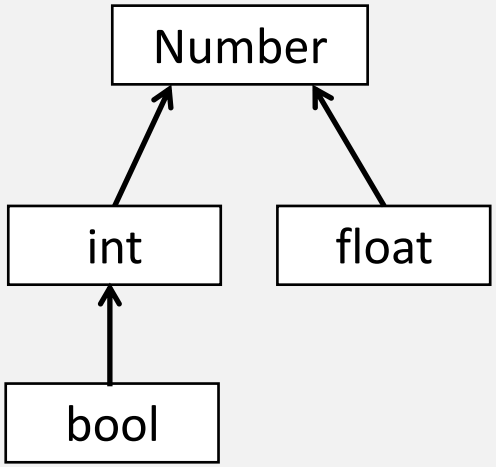
second or-operand only evaluated if first is *false*

```
def unaffordable(price, income):
    return income == 0 or price / income > 0.5
```

```
>>> 1 / 0 > 0.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> unaffordable(1, 0)
True
```

Boolean values have numeric interpretations

```
>>> True + 1
2
>>> False + False + False
0
>>> int(True)
1
>>> int(False)
0
>>> isinstance(True, int)
True
>>> isinstance(False, int)
True
```



Why is this useful?

Example: incrementing counter based on truth check

```
def count(x, seq):
    res = 0
    for e in seq:
        if e == x:
            res += 1
    return res
```

```
def count(x, seq):
    res = 0
    for e in seq:
        res += e==x
    return res
```

All objects can be interpreted as Boolean values

```
>>> 'apple pie' or 3.14159
'apple pie'
>>>
```

Note that values are interpreted as Booleans but not converted

```
>>> bool('apple pie') or bool(3.14159)
True
```

…most objects are interpreted as *True*

```
>>> bool(1)
True
>>> bool(3.14159)
True
>>> bool('apple pie')
True
>>>
```

…except a few cases of 'falsy' objects

```
>>> bool(1)
True
>>> bool(3.14159)
True
>>> bool('apple pie')
True
>>> bool(0)
False
>>> bool(0.0)
False
>>>
```

Numerical zero values

…except a few cases of 'falsy' objects

```
>>> bool(1)
True
>>> bool(3.14159)
True
>>> bool('apple pie')
True
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool([])
False
>>> bool(set())
False
>>>
```

empty collections

…except a few cases of 'falsy' objects

```
>>> bool(1)
True
>>> bool(3.14159)
True
>>> bool('apple pie')
True
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool([])
False
>>> bool(set())
False
>>> bool(None)
False
```

the None value

Why is this useful?

Example 1: interpreting adjacency matrix entries directly

```
def neighbours(i, graph):
    res = []
    for j in range(len(graph)):
        if graph[i][j]==1:
            res += [j]
    return res
```

```
def neighbours(i, graph):
    res = []
    for j in range(len(graph)):
        if graph[i][j]:
            res += [j]
    return res
```

Example 2: looping while collection not empty

```
def reachable(graph, s):
    visited = []
    boundary = [s]
    while len(boundary) > 0:
        v = boundary.pop()
        visited += [v]
        for w in neighbours(v, graph):
            if w not in visited and w not in boundary:
                boundary.append(w)
    return visited
```

```
def reachable(graph, s):
    visited = []
    boundary = [s]
    while boundary:
        v = boundary.pop()
        visited += [v]
        for w in neighbours(v, graph):
            if w not in visited and w not in boundary:
                boundary.append(w)
    return visited
```

Example 3: default values

```
def reachable(graph, s, visited=None):
    if visited is None: visited = [False]*len(graph)
    res = [s]
    visited[s] = True
    for v in neighbours(s, graph):
        if not visited[v]:
            res += reachable(graph, v, visited)
    return res
```

Uses the short-circuiting behaviour of or

```
def reachable(graph, s, visited=None):
    visited = visited or [False]*len(graph)
    res = [s]
    visited[s] = True
    for v in neighbours(s, graph):
        if not visited[v]:
            res += reachable(graph, v, visited)
    return res
```

Feels idiosyncratic but very common in the JavaScript (Web) world

Iterable object *comprehensions*

How to obtain column of table?

transaction	date	mail_address	product	price
1	1-2-2019	joe@kmail.com	MultiQuick 3	113
2	1-2-2019	patricia@cmail.com	NutriBullet 900	85
3	2-2-2019	joe@kmail.com	NutriNinja	88
4	2-2-2019	elenor@amail.com	NutriNinja	80
5	2-2-2019	joe@kmail.com	NutrilBullet1200	136
…	…	…	…	…
1000	22-05-2019	ramio@manosh.edu	Prospero	149

While loop version

```
def col(j, table):
    res = []
    i = 0
    while i < len(table):
        res += [table[i][j]]
        i += 1
    return res
```

For loop version

```
def col(j, table):
    res = []
    for i in range(len(table)):
        res += [table[i][j]]
    return res
```

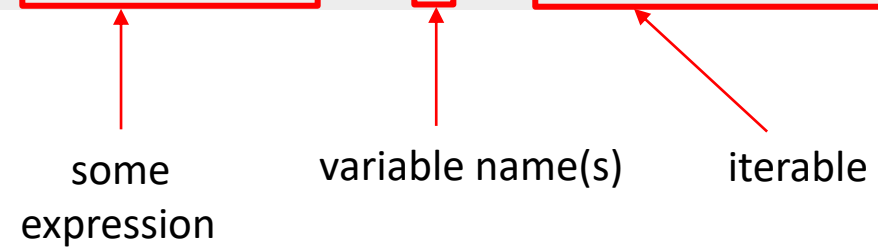
direct declaration of relevant indices

How about direct declaration of result list?

```
def col(j, table):
    return [[table[i][j]] for i in range(len(table))]
```

List comprehensions allow to directly “declare” list

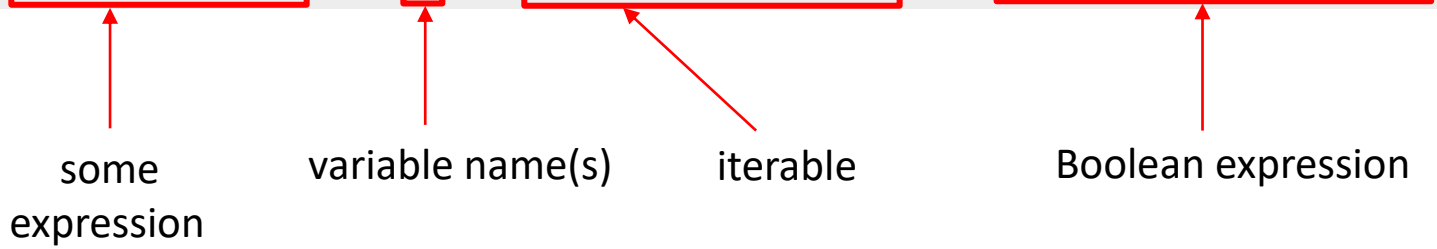
```
[[table[i][j]] for i in range(len(table))]
```




```
>>> [i/2 for i in range(10)]
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
```

...can also involve filter

```
[[table[i][j]] for i in range(len(table)) if table[i][k]=='joe']
```




```
>>> [i/2 for i in range(10)]
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
```

```
>>> [i/2 for i in range(10) if not (i/2).is_integer()]
[0.5, 1.5, 2.5, 3.5, 4.5]
```

Can replace map function

```
>>> quantities =
list_from_file('quantities.txt')
>>> quantities
['300', '300', '200', '100', '250', '100',
'120', '200']
>>> list(map(int, quantities))
[300, 300, 200, 100, 250, 100, 120, 200]
>>> [int(s) for s in quantities]
[300, 300, 200, 100, 250, 100, 120, 200]
```

…but is more flexible

```
>>> quantities = list_from_file('quantities.txt')
>>> quantities
['300', '300', '200', '', '250', '100', '120', '200']
>>> list(map(int, quantities))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: ''
>>> def digit(s): return s.isdigit()
>>> list(filter(digit, quantities))
['300', '300', '200', '250', '100', '120', '200']
>>> list(map(int, filter(digit, quantities)))
[300, 300, 200, 250, 100, 120, 200]
```

```
>>> [int(s) for s in quantities if s.isdigit()]
[300, 300, 200, 100, 250, 100, 120, 200]
```

Recall anagram algorithm

words
solver
camera
waster
typhon
tawers
lovers
waters
higher
python
rawest

transform

augment by *order-invariant* key

key	words
elorsv	solver
aacemr	camera
aerstw	waster
hnopty	typhon
aerstw	tawers
elorsv	lovers
aerstw	waters
eghhir	higher
hnopty	python
aerstw	rawest

conquer

group by key

key	words
aacemr	camera
aerstw	rawest
aerstw	tawers
aerstw	waster
aerstw	waters
eghhir	higher
elorsv	lovers
elorsv	solver
hnopty	python
hnopty	typhon

Recall anagram function

```
def anagrams(words):
    augmented = order_invariant_indexing(words)
    ordered = sorted(augmented)

    res = []
    i = 0
    while i < len(ordered):
        res.append([])
        key = ordered[i][0]
        while i < len(ordered) and \
                ordered[i][0] == key:
            res[-1].append(ordered[i][1])
            i = i + 1
    return res
```

```
>>> words = ['solver', 'camera', 'lovers', …,'rawest']
>>> sorted(order_invariant_indexing(words))
[('aacemr', 'camera'), ('aerstw', 'rawest'), …,
('hnopty', 'python'), ('hnopty', 'typhon')]
```

Recall anagram function

```
>>> sorted('waters')
['a', 'e', 'r', 's', 't', 'w']
>>> ''.join(sorted('waters'))
'aerstw'
```

```
def order_invariant_indexing(words):
    res = []
    for word in words:
        res += [(''.join(sorted(word)), word)]
    return res
```

```
def order_invariant_indexing(words):
    return [(''.join(sorted(word)), word) for word in
words]
```

Example from nutrition app

```
def scaled(row, alpha):
    """
    Input : list with numeric entries (row), scaling factor
(alpha)
    Output: new list (res) of same length with
res[i]==row[i]*alpha

    For example:
    >>> scaled([1, 4, -1], 2.5)
    [2.5, 10.0, -2.5]
    """
    res = []
    for x in row:
        res += [alpha*x]
    return res
```

```
def scaled(row, alpha):
    return [alpha*x for x in row]
```

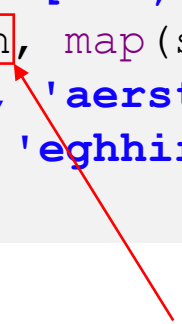
Example from nutrition app

```
def sum_of_rows(r1, r2):
    """
    Input : two lists (r1, r2) with same
number of numeric entries
    Output: new list (res) of same length
with res[i]==r1[i]+r2[i]
            for all i in range(len(r1))

    For example:
    >>> sum_of_rows([100, -4, 10], [0, 3.5,
-10])
    [100, -0.5, 0]
    """
    return [r1[j]+r2[j] for j in range(n)]
```

Higher-order and nested functions

Recall that methods are functions "bound" to a specific object

```
>>> ''.join
<built-in method join of str object at 0x1066593b0>
>>> list(map(sorted, words))
[['e', 'l', 'o', 'r', 's', 'v'], ['a', 'a', 'c', 'e',
'm', 'r'], ['a', 'e', 'r', 's', 't', 'w'], ['h', 'n',
'o', 'p', 't', 'y'], ['a', 'e', 'r', 's', 't', 'w'],
['e', 'l', 'o', 'r', 's', 'v'], ['a', 'e', 'r', 's',
't', 'w'], ['e', 'g', 'h', 'h', 'i', 'r'], ['h', 'n',
'o', 'p', 't', 'y'], ['a', 'e', 'r', 's', 't', 'w']]
>>> list(map(''.join, map(sorted, words)))
['elorsv', 'aacemr', 'aerstw', 'hnopty', 'aerstw',
'elorsv', 'aerstw', 'eghhir', 'hnopty', 'aerstw']
>>>
```



can use method as argument just like any other function

Sometimes we would like to *unbind* method from specific object

```
>>> a, b = '1', 'two'
>>> a.isdigit
<built-in method isdigit of str object at 0x10db843b0>
>>> f = a.isdigit
>>> f()
True
>>> f(b)
TypeError: isdigit() takes no arguments (1 given)
>>> str.isdigit
<method 'isdigit' of 'str' objects>
>>> f=str.isdigit
>>> f(a)
True
>>> f(b)
False
>>>
```

general function that takes string object as first argument

Application: filtering digits

```
>>> string = 'successfully read 14 files'
>>> def isdigit(c): return c.isdigit()
>>> ''.join(filter(isdigit, string))
'14'
```

```
>>> string = 'successfully read 14 files'
>>> ''.join(filter(str.isdigit, string))
'14'
```

```
>>> string = 'successfully read 14 files'
>>> ''.join([c for c in string if c.isdigit()])
'14'
```

Application: unifying graph traversals

```
def bfs_traversal(graph, s):
    visited = []
    boundary = deque([s])
    while len(boundary) > 0:
        v = boundary.popleft()
        visited.append(v)
        for w in neighbours(v, graph):
            if w not in visited and w not in
boundary:
                boundary.append(w)
    return visited
```

differ only in one method call

```
def dfs_traversal(graph, s):
    visited = []
    boundary = deque([s])
    while len(boundary) > 0:
        v = boundary.pop()
        visited.append(v)
        for w in neighbours(v, graph):
            if w not in visited and w not in boundary:
                boundary.append(w)
    return visited
```

Application: unifying graph traversals

```
def traversal(graph, s, popnext):
    visited = []
    boundary = deque([s])
    while len(boundary) > 0:
        v = popnext(boundary)
        visited.append(v)
        for w in neighbours(v, graph):
            if w not in visited and w not in boundary:
                boundary.append(w)
    return visited
```

```
>>> traversal(graph, s, deque.popleft)
000---001---002   008---013   019---026   033---038---042
              |     |           |           |           |
007---004---003---005---009---014---020---027---034   045
              |                 |     |     |     |     |
015---010---006---011---017   021   028   035   039   048
 |     |     |                       |           |
022   016   012---018---024---031   036---040   043---046
 |     |           |     |
029   023---030   025   032---037---041---044---047---049
```

Application: unifying graph traversals

```
def traversal(graph, s, popnext):
    visited = []
    boundary = deque([s])
    while len(boundary) > 0:
        v = popnext(boundary)
        visited.append(v)
        for w in neighbours(v, graph):
            if w not in visited and w not in boundary:
                boundary.append(w)
    return visited
```

```
>>> traversal(graph, s, deque.pop)
000---001---002   046---047   044---045   039---040---041
              |     |           |           |           |
049---048---003---025---026---027---029---033---035   042
              |                 |     |     |     |     |
022---018---004---016---017   028   030   034   036   043
 |     |     |                       |           |
023   019   005---006---008---015   031---032   037---038
 |     |           |     |
024   020---021   007   009---010---011---012---013---014
```

Nested Functions

```
def fun1(a):

    def fun2(b):
        return a + b

    return fun2(1)
```

nested function “sees” variables of outer function

```
>>> fun1(1)
2
>>> fun1(-1)
0
```

Application: “function factories”

```
def func(c):

    n = len(c)

    def p(x):
        return sum(c[i]*x**i for i in range(n))

    return p
```

```
>>> f = func([0, -1, 0, 1])
>>> f(2)
?
```

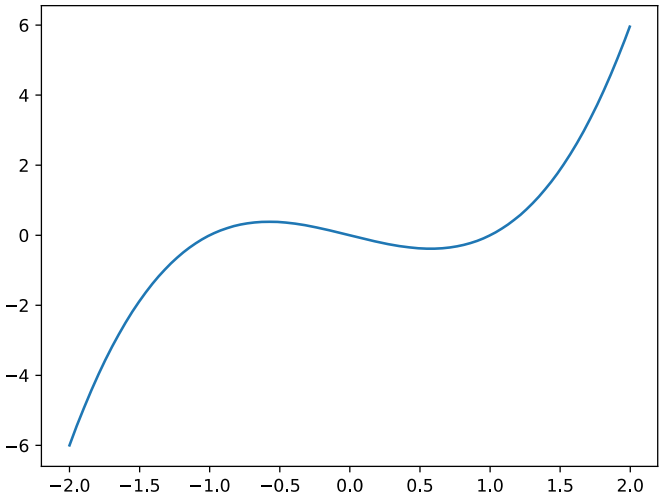
1. Visit https://flux.qa
2. Log in using your Authcate details (not required if you’re already logged in to Monash)
3. Touch the + symbol and enter the code: HHYBXU
4. Answer questions when they pop up.

Application: “function factories”

```
def polynomial(coeffs):

    n = len(coeffs)

    def p(x):
        return sum(coeffs[i]*x**i for i in range(n))

    return p
```

$f(x) = x^3 - x$

```
>>> f = polynomial([0, -1, 0, 1])
>>> f(1)
0
>>> f(2)
6
```



Recall our nutrition table

	energy	water	protein	carbs	sugars	fat	fibres
apple	229	84.3	0.4	12.0	11.8	0.0	2.3
orange	186	84.3	1	9.5	8.3	0.2	2.1
broccoli	124	89.6	3.2	2.0	2.0	0.1	4.1
beef	613	70	22.8	0.2	0.0	6.0	0.0
lamb	1057	60.2	18.6	0.0	0.0	20.2	0.0
bread	1446	37.6	8.4	43.5	1.5	2.6	6.9

… and its representation in Python

```
cols = ['energy', …,'carbs', 'sugars', 'fat', 'fibres']
rows = ['apple', …,'beef', 'lamb', 'bread']

nutr_vals = [[229, 84.3, 0.4, 12.0, 11.8, 0.0, 2.3],
             [186, 84.3, 1, 9.5, 8.3, 0.2, 2.1],
                                …
             [1446, 37.6, 8.4, 43.5, 1.5, 2.6, 6.9]]
```

Assume we want to ***rank food*** in terms of one of its attributes

…built-in function `sorted` can help us with that

(with some basic understanding of higher order functions!)

Recall our nutrition table

```
def food_ranking_by(attr):
    m = len(nutr_vals)
    j = cols.index(attr)

    ranks = sorted(range(m), key=attr_value)

    res = []
    for row in ranks:
        res.append(rows[row])

    return res
```

function argument that determines sorting score

returns input iterable sorted according to score

Recall our nutrition table

```
def food_ranking_by(attr):
    m = len(nutr_vals)
    j = cols.index(attr)

    def attr_value(i):
        return nutr_vals[i][j]

    ranks = sorted(range(m), key=attr_value)

    res = []
    for row in ranks:
        res.append(rows[row])

    return res
```

define as nested function

function argument that determines sorting score

returns input iterable sorted according to score

```
>>> food_ranking_by('energy')
['broccoli', 'orange', 'apple', 'beef', 'lamb', 'bread']
>>> food_ranking_by('fat')
['apple', 'broccoli', 'orange', 'bread', 'beef', 'lamb']
```

Summary

1. Boolean values and expressions
   - Short-circuiting
   - Booleans are integers
   - Boolean interpretations of objects
2. Iterable object *comprehensions*
3. Higher-order and nested functions
   - Unbinding methods as functions
   - Nested functions, e.g., to use as return values

Coming Up

- Gaussian elimination
- Combinatorial Optimisation