Discrete Mathematics

Topic 01 : Computational Thinking

Lecture 09 : Selection of Python Tasks

Dr Kieran Murphy

Computing and Mathematics, SETU (Waterford).
(kieran.murphy@setu.ie)

Autumn Semester, 2025/26

Outline

- Different coding styles — using a result variable or multiple print statements
- Different use cases — code block or functions
- Topic specific tasks — Logic, Sets, Collections, Relations, Functions, Enumeration, . . .

Outline

Motivation

These notes* are an attempt to collate all of the Python tasks that we have discussed during the module, as well some comments on general coding patterns and processes.

To remind you of how we integrate Python into the *Discrete Mathematics* module:

- In the tutorials (online quizzes):
  - Multiple choice questions where you are asked to identify the output (or the purpose of) of some code.
- In the practicals:
  - You are asked to either use existing methods (e.g. set intersection, union, etc.) or implement your own using loops, conditional statements and add/append operations.
- In the end of semester examination:
  - You should expect to be asked to read and execute python code, NOT write it.
  - Python interpretation questions will be similar to those given in the online multiple choice quizzes (with the multiple choices options removed) or similar to those you were asked to write in the practicals.

*This is a work in progress, so expect corrections/changes!

Python Cheat Sheet

Base Types

integer, float, boolean, string, bytes

int	163 0 −192 $\underbrace{\text{0b110}}_{\text{binary}}$ $\underbrace{\text{0x3F}}_{\text{hex}}$
float	9.32 0.0 $-1.7\underbrace{\text{E}-6}_{\times 10^{-6}}$
bool	$\underbrace{\text{False}}_{0}$ $\underbrace{\text{True}}_{1}$
str	'some text' or "some text"
bytes	b"text\xfe\775"

Container Types

ordered containers — repeatable values

list	[1,5,3]	["a",1,5,5]	[5]	[]
tuple	(1,5,3)	"a",1,5,5	(5,)	()
str	"153"			""

Immutable (non-modifiable values)

key containers — no order, unique keys

set	{"key1","key2"}	{1,9,3,0}	**set**()
dict	{"key1":value1,"key2":value2}	**dict**(a=3,b="v")	{}

Integer Sequences

range([start,] end [,step])

start default is 0 (inclusive), end (exclusive), step default is 1.

range(5) → 0,1,2,3,4
range(2,5) → 2,3,4
range(2,12,3) → 2,5,8,11
range(20,5,−5) → 20,15,10

Operations on Sets

Operators

|	.union
&	.intersection
−	.difference

Methods

s.add(key)	s.update(s2)
s.clear()	s.remove(key)

Operations on Lists

Methods

a.append(value)	a.extend(a2)
s.insert(idx,value)	a.pop()

Conversions

type(expression)

int('153') → 15	
int('3f',16) → 63	(Specify base in 2[nd] parameter)
int(−11.24e8) → −1124000000	
int(15.56) → 15	(Truncate decimal point)
round(15.58,1) → 15.6	(Round to 1 decimal place)
float('15.56') → 15.56	
bool(x)	(False for None, zero or empty containers)
str(x)	(String representation of x.)

chr(65) → 'A'   code ↔ char   **ord**('A') → 65

list('abc') → ['a','b','c']
dict([(3,'three'), (1,'one')]) → {3:'three', 1:'one'}
set(['one','two']) → {'one','two'}

(Split string using a separator, **str** → **list** of **str**)
'random:data:666'.split(':') → ['random', 'data', '666']

(Join a list of strings, **list** of **str** → **str**)
':'.join(['random', 'data', '666']) → 'random:data:666'

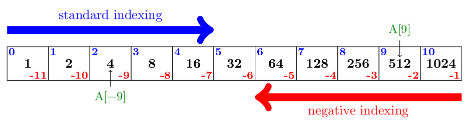
(Convert each element in a collection)
[**int**(x) **for** x **in** ['1','29','−3']] → [1, 29, −3]

Generic Operations on Containers

min(c) **max**(c) **sum**(c) **sorted**(c)
len(c) (Number of elements in collection c)

all(c) → True if **all** items in c evaluate to True, else False.
any(c) → True if **at least one** item in c evaluate to True, else False.

Sequence Containers Indexing



a[3:6] → [8, 16, 32]
a[1:−1] → [2, 4, 8, 16, 32, 64, 128, 256, 512]
a[::−1] → [1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1]

Looping over Collections

(Loop over values)
```
for value in A:
    print(value)
```

(Count and loop over values)
```
for k,value in enumerate(A):
    print(k, value)
```

(While loop)
```
k = 0
while k<len(A):
    print(k, A[k])
    k += 1
```

Initialisation **before** loop.
update **within** loop.

break immediatly exits loop. **continue** skips to next iteration.
else block for normal loop exit.

A copy of this will be attached to your exam paper.

Testing Quantifiers

$\forall x (x \in A) \, [x \text{ contains the digit } 3]$

universal quantifier

- To prove **True**, you need to check proposition for every element in A.
- To prove **False**, you just find one element for which the proposition is **False**.

```
all(prop(x) for x in A)
```

"x contains the digit 3"

This is a proposition.
It may be **True** or **False**, depending on the choice for x.

E.g., ff $x = 32$ then it is **True**, if $x = 16$ then it is **False**.

existance quantifier

We could implement this as the function:

```
def prop(x):
  return '3' in str(x)
```

$\exists x (x \in A) \, [x \text{ contains the digit } 3]$

- To prove **True**, you just find one element for which the proposition is **True**.
- To prove **False**, you need to check proposition for every element in A.

```
any(prop(x) for x in A)
```

Different Coding Styles — Single vs Multiple Output Statements

Consider the task of testing if two sets, A and B, are **disjoint**†

Algorithm: For each element in A, check that it is in B. If the element is in B then the sets are not disjoint and we stop searching.

Using single output

```
A = {2,3,5,7}
B = {4,5,6}

result = True

for x in A:
    if x in B:
        result = False
        break

print(result)
```

- Single **print** statement.
- So need variable to store desired output.

- Multiple **print** statements.
- No variable needed to store output.
- But need a **loop else** block (only runs if loop ends normally=no break).

Using multiple outputs

```
A = {2,3,5,7}
B = {4,5,6}

for x in A:
    if x in B:
        print(False)
        break
else:
    print(True)
```

† disjoint sets have no elements in common, i.e., their intersection is the empty set.

Different Use Cases — Code Block or Functions (Single Output/Return)

The "Using single output" version when converted to a function has a single **return** statement.

Using single output

```python
A = {2,3,5,7}
B = {4,5,6}

result = True

for x in A:
    if x in B:
        result = False
        break

print(result)
```

- Single **print** statement.
- So need variable to store desired output.

- Single **return** statement.
- No variable needed to store output.
- Function body is nearly identical to original code block.

Using single return

```python
def is_disjoint(A, B):

    result = True

    for x in A:
        if x in B:
            result = False
            break

    return result


A = {2,3,5,7}
B = {4,5,6}
print(is_disjoint(A,B))
```

Different Use Cases — Code Block or Function (Multiple Output/Return)

The "Using multiple outputs" version when converted to a function has a multiple **return** statements.

Using multiple output

```python
A = {2,3,5,7}
B = {4,5,6}

for x in A:
    if x in B:
        print(False)
        break
else:
    print(True)
```

- Multiple **print** statement.
- So need **loop else** block.

- Multiple **return** statements.
- No need for **break** statement due to **return**.
- The **loop else** block is simplified due to **return**.

Multiple **return** style of function tends to result in shorter functions with less nesting.

Using multiple return

```python
def is_disjoint(A, B):

    for x in A:
        if x in B:
            return False

    return True


A = {2,3,5,7}
B = {4,5,6}
print(is_disjoint(A,B))
```

Selection of Tasks

In the following slides I have collated most of the various tasks that have appeared either in the notes or as questions in the quizzes or the practicals. To keep the number of slides down I only give the "function with multiple return statements" version — this is the coding style that I personally prefer, and use, but be prepared to see the tasks given in the other styles also.

Also, they can often be multiple logically and computationally equivalent solutions to these tasks. So any code presented in the end of semester exams may vary. However, rest assured, I will not intentionally try to obfuscate code, so if you understand the python control statements you should be able to identify the task.

Also, tasks can be combined. For example, combing relation task `is_injective` with function task `is_function` to get code that will check if a given relation is an injective function.

Outline

Set Tasks

Properties

- is equal
- is disjoint
- is subset
- is proper subset

Operations

- intersection
- union
- set difference
- symmetric difference
- Cartesian product

Set Task — `is_equal` $A = B$

TASK

Given two sets, A and B, determine if they are equal*.

ALGORITHM

STEP 1 If size of A is not equal to size of B then sets are not equal.

STEP 2 Else for all elements in A, if element not in B then sets are not equal.

STEP 3 Else sets are equal.

```python
def is_equal(A, B):

    if len(A)!=len(B): return False

    for x in A:
        if x not in B:
            return False

    return True
```

```python
def is_equal(A, B):

    if len(A)!=len(B): return False

    return all(x in B for x in A)
```

Step 2 is based on the predicate $\forall x (x \in A)\,[x \in B]$. We can implement this in Python using **`any`**.

*Two sets are equal iff they have the same elements.

Set Task — `is_disjoint`

$A \cap B = \emptyset$

TASK

Given two sets, A and B, determine if they are disjoiint*.

ALGORITHM

STEP 1 For all elements in A, if element in B then sets are not disjoint.

STEP 2 Else sets are disjoint.

9

```python
def is_disjoint(A, B):

    for x in A:
        if x in B:
            return False

    return True
```

10

```python
def is_disjoint(A, B):

    return all(x not in B for x in A)
```

Step 1 is based on the predicate $\forall x (x \in A) [x \notin B]$. We can implement this in Python using **any**.

*Two sets are **disjoint** iff they have no elements in common, i.e., intersection is zero.

Set Task — `is_subset` $A \subseteq B$

Task

Given two sets, A and B, determine if A is a subset* of B.

Algorithm

Step 1 For all elements in A, if element not in B then A is not a subset of B.

Step 2 Else A is a subset of B.

11
```python
def is_subset(A, B):

    for x in A:
        if x not in B:
            return False

    return True
```

Step 1 is based on the predicate $\forall x(x \in A)\,[x \in B]$.
We can implement this in Python using **any**.

12
```python
def is_subset(A, B):

    return all(x in B for x in A)
```

*A is a **subset** of B iff every element in A is also in B.

Set Task — `is_proper_subset` $A \subset B$

TASK

Given two sets, A and B, determine if A is a proper subset* of B.

ALGORITHM

STEP 1 For all elements in A, if element not in B then A is not a subset of B.

STEP 2 Else A is a proper subset of B, iff A is smaller than B.

13

```python
def is_proper_subset(A, B):

    for x in A:
        if x not in B:
            return False

    return len(A)<len(B)
```

14

```python
def is_proper_subset(A, B):

    return all(x in B for x in A) and len(A)<len(B)
```

*A is a **proper subset** of B iff every element in A is also in B and B contains at least one element that is not in A.

Set Task — `intersection` $A \cap B$

TASK

Given two sets, A and B, construct their intersection*, i.e., the set $A \cap B$.

ALGORITHM

STEP 1 Create empty set C.

STEP 2 For all elements in A, if element in B then add element to C.

STEP 3 Return C.

15

```python
def intersection(A, B):

    C = set()

    for x in A:
        if x in B:
            C.add(x)

    return C
```

…or using list comprehension …

16

```python
def intersection(A, B):

    return {x for x in A if x in B}
```

*$A \cap B$ is the set of all elements that are in both A and in B.

Set Task — `union` $A \cup B$

TASK

Given two sets, A and B, construct their union*, i.e., the set $A \cup B$.

ALGORITHM

STEP 1 Create empty set C.

STEP 2 For all elements in A, add element to C.

STEP 3 For all elements in B, add element to C.

STEP 4 Return C.

```python
def union(A, B):

    C = set()

    for x in A:
        C.add(x)

    for x in B:
        C.add(x)

    return C
```

We could combine steps 1 and 2 by setting C to be a copy of A using code

```python
C = A.copy()
```

Note using `C=A` does not create a separate set!

* $A \cup B$ is the set of all elements that are in A or in B or in both.

Set Task — set_difference $A \setminus B$

Task

Given two sets, A and B, construct the set difference*, i.e., the set $A \setminus B$.

Algorithm

Step 1 Create empty set C.

Step 2 For all elements in A, if element not in B then add element to C.

Step 3 Return C.

```python
def set_difference(A, B):

    C = set()

    for x in A:
        if x not in B:
            C.add(x)

    return C
```

```python
def set_difference(A, B):

    return {x for x in A if x not in B}
```

*$A \setminus B$ is the set of all elements that are in A but are not in B.

Set Task — `symmetric_difference` $A \oplus B$

TASK

Given two sets, A and B, construct the symmetric difference*, i.e., the set $A \oplus B$.

ALGORITHM

STEP 1 Create empty set C.

STEP 2 For all elements in A, if element not in B then add element to C.

STEP 3 For all elements in B, if element not in A then add element to C.

STEP 4 Return C.

```python
def symmetric_difference(A, B):

    C = set()

    for x in A:
        if x not in B:
            C.add(x)

    for x in B:
        if x not in A:
            C.add(x)

    return C
```

*$A \oplus B$ is the set of all elements that are in A or in B but not in both sets.

Set Task — `cartesian_product` $A \times B$

TASK

Given two sets, A and B, construct the Cartesian product*, i.e., the set $A \times B$.

ALGORITHM

STEP 1 Create empty set C.

STEP 2 For all elements a in A, for all elements b in B add ordered pair (a, b) to C.

STEP 3 Return C.

21

```python
def cartesian_product(A, B):

    C = set()

    for a in A:
        for b in B:
            C.add( (a,b) )

    return C
```

...or using list comprehension ...

22

```python
def cartesian_product(A, B):

    C = { (a,b) for a in A for b in B }

    return C
```

*$A \times B$ is the set of all ordered pairs (a, b) where a is an element of A and b is an element of B.

Outline

Relation Tasks

Properties

- is a relation from set A to set B
- is reflexive
- is symmetric
- is transitive
- is irreflexive
- is anti-symmetric
- is asymmetric
- is surjective (onto) (remember if a relation is not onto then it is into)
- is injective (one-to-one)

Relation Task — `is_relation`

TASK

Given relation, R and two sets, A and B, determine if R is a relation* from A to B.

ALGORITHM

STEP 1 For all pairs (a, b) in R, if element a is not in A, then result is `False`.

STEP 2 For all pairs (a, b) in R, if element b is not in B, then result is `False`.

STEP 3 Else R is a relation from A to B.

```
def is_relation(R, A, B):

    for (a,b) in R:
        if a not in A:
            return False
        if b not in B:
            return False

    return True
```

The loops in step 1 and step 2 can be merged (as is done in code above).

*A **relation from** A **to** B is a set of ordered pairs, (a, b), where a is an element of A and b is an element of B.

Relation Task — `is_reflexive`

TASK

Given a relation R on a set A, determine if R is reflexive*.

ALGORITHM

STEP 1 For all a in A, if ordered pair (a, a) is not in R, then R is not reflexive.

STEP 2 Else R is reflexive.

Note that we need the set A to test if R is reflexive.

```python
def is_reflexive(R, A):

    for a in A:
        if (a,a) not in R:
            return False

    return True
```

An alternative approach is to build a set of all pairs representing the self-loops, and then checking if that set is a subset of R.

Or can use the python `all` predicate.

```python
def is_reflexive(R, A):

    return all( (a,a) in R for a in A )
```

*A relation, R, on A is **reflexive** if all ordered pairs, (a, a) are in R where a is an element of A.

Relation Task — `is_symmetric`

TASK

Given a relation R on a set A, determine if R is symmetric*.

ALGORITHM

STEP 1 For all (a, b) in R, if ordered pair (b, a) is not in R, then R is not symmetric.

STEP 2 Else R is symmetric.

26

```
def is_symmetric(R):

    for (a,b) in R:
        if (b,a) not in R:
            return False

    return True
```

*A relation, R, on A is **symmetric** if ordered pair (b, a) is in R whenever (a, b) is in R.

Relation Task — `is_transitive`

TASK

Given a relation R on a set A, determine if R is transitive*.

ALGORITHM

STEP 1 For all two-hop paths in R, if the corresponding one-hop path is not in R, then R is not transitive.

STEP 2 Else R is transitive.

27

```
def is_transitive(R):

    for (a,b) in R:
        for (c,d) in R:
            if b==c and (a,d) not in R:
                return False

    return True
```

To find all two-hop paths in R we use a nested for loop. Outer loop finds all pairs (a, b), inner for loops finds all pairs (c, d).
Whenever $b = c$ we have a two-hop path $(a \to b = c \to d)$.

Then if one-hop path (a, d) is not in R, R is not transitive.

*A relation, R, on A is **transtive** iff, whenever (a, b) is in R and (b, c) is in R then (a, c) is in R.

Relation Task — `is_irreflexive`

TASK

Given a relation R on a set A, determine if R is irreflexive*.

ALGORITHM

STEP 1 For all a in A, if ordered pair (a, a) is in R, then R is not irreflexive.

STEP 2 Else R is irreflexive.

28

```python
def is_irreflexive(R, A):

    for a in A:
        if (a,a) in R:
            return False

    return True
```

An alternative approach is to build a set of all pairs representing the self-loops, and then checking if that set and R are disjoint.

Or can use the pyth on **any** predicate.

29

```python
def is_irreflexive(R, A):

    return not any( (a,a) in R for a in A )
```

*A relation, R, on A is **irreflexive** if R does not contain any ordered pairs like (a, a) where a is an element of A.

Relation Task — `is_antisymmetric`

TASK

Given a relation R on a set A, determine if R is anti-symmetric*.

ALGORITHM

STEP 1 For all pairs (a, b) R, if (b, a) is also in R and $a \neq b$ then R is not anti-symmetric.

STEP 2 Else R is anti-symmetric.

```python
def is_antisymmetric(R):

    for (a,b) in R:
        if (b,a) in R and a!=b:
            return False

    return True
```

*A relation, R, on A is **antisymmetric** iff whenever both (a, b) and (b, a) are in R then $a = b$.

Relation Task — `is_asymmetric`

TASK

Given a relation R on a set A, determine if R is asymmetric*.

ALGORITHM

STEP 1 For all pairs (a, b) R, if (b, a) is also in R then R is not asymmetric.

STEP 2 Else R is asymmetric.

```python
def is_asymmetric(R):

    for (a,b) in R:
        if (b,a) in R:
            return False

    return True
```

*A relation, R, on A is **asymmetric** iff whenever (a, b) is in R then (b, a) is not in R (including case $a = b$).

Relation Task — `is_surjective`

TASK

Given a relation R from set A to set B, determine if R is surjective*.

ALGORITHM

STEP 1 Build a set of the second elements in the each of ordered pairs of R.

STEP 2 If this set equals the target (B) then R is surjective, else it is not surjective.

```python
def is_surjective(R,B):

    image = { b for (_,b) in R }

    return image==B
```

*A relation, R, from A to B is **surjective (onto)** iff the image of R equals the target (here target is B). In other words every element in B occurs, at least once, as the second element in the ordered pairs of R.

Relation Task — `is_injective`

TASK

Given a relation R from set A to set B, determine if R is injective*.

ALGORITHM

STEP 1 Create empty set to store the image of R.

STEP 2 For each ordered pair (a, b) in R if b in `image` then R is not injective and STOP, else add b to `image`.

STEP 3 Else R is injective.

33

```python
def is_injective(R):

    image = set()

    for (_,b) in R:
        if b in image:
            return False
        image.add(b)

    return True
```

A common trick in python to check if there are repeated items in a collection is to convert to a set and compare size.

34

```python
def is_injective(R):

    tmp = [b for (_,b) in R]

    return len(tmp)==len(set(tmp))
```

*A relation, R, from A to B is **injective (one-to-one)** iff different first elements in the ordered pairs of R implies different second elements in the ordered pairs. In other words every element in B occurs, at most once, as the second element in the ordered pairs of R.

Outline

Function Tasks

Properties

- is given relation from set A to set B a function ?

Function Task — `is_function`

TASK

Given a function R from set A to set B, determine if R is a function*.

ALGORITHM

STEP 1 Create empty set, D, to store the domain of R.

STEP 2 For each ordered pair (a, b) in R if a in D then R is not a function and STOP, else add a to D.

STEP 3 Else R is function iff D equals A.

35

```python
def is_function(R, A):

    D = set()
    for (a,_) in R:
        if a in D:
            return False
        D.add(a)

    return D==A
```

36

```python
def is_function(R, A):

    tmp = [ a for (a,_) in R ]
    D = set(tmp)

    return len(tmp)==len(D) and D==A
```

Notice that step 2 is testing for "at most once", while step 3 is testing for "at least once".

*A relation, R, from A to B is a **function** iff every element in A appears exactly once as a first element in the ordered pairs of R. In other words every element in A occurs at most once and at least once.