Lecture 6: Relations and functions

CAB203 Discrete Structures

Matthew McKague

Queensland University of Technology

matthew.mckague@qut.edu.au

Outline

Readings

This week:

- Pace: 5.1 to 5.3, 6.1.4, 6.2, 7.2
- Lawson: 3.3 to 3.6

Next week:

- Pace: 7.3

Outline

Relations

Functions

Sequences

Tuples

The notation (a, b) is a *ordered pair*, and the order matters.
Sets of two:

- $\{a, b\}$ is a set with elements a and b
- $\{a, b\} = \{b, a\}$
- $\{a, a\} = \{a\}$

Pairs:

- $(a, b) \neq (b, a)$ unless $a = b$
- $(a, a) \neq (a)$

More generally, we have $(a_1, \ldots, a_n)$ is an n-tuple: n elements, where the order matters.

Tuples examples

- $(1, 2)$
- $(2, 2)$
- (cat, dog)
- $(\text{“John”}, \text{“Smith”}, 36)$

Formally, an ordered pair is usually defined by the Kuratowski definition: $(x, y) = \{\{x\}, \{x, y\}\}$.

Cartesian product

Given sets A and B we define the *Cartesian product* to be the set

$$A \times B = \{(a, b) : a \in A, b \in B\}$$

The size of $A \times B$ is given by $|A \times B| = |A||B|$.

More Cartesian products

More generally we have:

- $A_1 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_1 \in A_1, \ldots, a_n \in A_n\}$
- $A^n = A \times A \times \cdots \times A$ (n copies of A)
- $|A_1 \times \cdots \times A_n| = |A_1| \cdots |A_n|$

Some examples

- $\mathbb{R}^2$ describes points on a 2-dimensional plane
- $\{0, 1\}^n$ is (equivalent to) the set of bit strings of length n
- $KEYS \times VALUES$ might describe the possible key-value pairs in a hash map
- $\{0, \ldots, 1919\} \times \{0, \ldots, 1079\}$ encodes (x, y)co-ordinates on a 1080p screen.

Tuples in Python

```
>>> t = (1, 2)        # Tupple literal
>>> t[0]              # accessing item
1
>>> t[0] = 1          # can't modify a tuple
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> s = (1,2,3,4,5,6) # tuples with many entries
```

Relations

Relations are a basic building block in mathematics:

- A *relation* on $A_1 \ldots A_n$ is a subset of $A_1 \times \cdots \times A_n$
- A *binary relation* between A and B is a subset of $A \times B$
- A binary relation between A and A is called a *relation over A*

We'll concentrate on binary relations.

Relation examples

- $\{(1,1),(2,2)\}$
- $\{(a,b) \in \mathbb{R}^2 : a = b\}$ (equality)
- $\{(a,b) \in \mathbb{Z}^2 : \exists c \in \mathbb{Z}\, a = bc\}$
- $\{(a,b) \in \mathbb{R}^2 : b = a^2\}$
- $\leq$, $<$, $=$, $\geq$, $>$
- The rows in a relational database
- Key-value pairs in an associative array (hash map)
- The (x,y)co-ordinates for drawing a happy face

If R is a binary relation then we write aRb to mean $(a,b) \in R$.
Hence $a \leq b$ is shorthand for $(a,b) \in \leq$.

Properties of relations

We can identify special properties that some binary relations will have

- symmetric
- reflexive
- transitive
- anti-symmetric
- irreflexive

We also identify special kinds of binary relations that have some of these properties

Symmetry

We say that a binary relation $R \subseteq A \times A$ is *symmetric* if

$$\forall (a, b) \in A \times A \ (aRb \leftrightarrow bRa)$$

That is, whenever we have (a, b) we also have (b, a).

Examples:

- $=$
- $a \equiv b \pmod{n}$ (equivalence modulo n)
- $\emptyset$, $A \times A$ (i.e. the trivial relations)

Anti-symmetry

A binary relation $R \subseteq A \times A$ is *anti-symmetric* if

$$\forall x, y \in A\ ((xRy \wedge yRx) \rightarrow x = y)$$

or, using the contrapositive

$$\forall x, y \in A\ (x \neq y \rightarrow \neg(xRy \wedge yRx))$$

In other words, if x and y are different then we can't have both xRy and yRx.

Examples:

- $<, >$
- $\leq, \geq$
- $\subseteq, \subset$

Reflexivity

We say that a binary relation $R \subseteq A \times A$ is *reflexive* if

$$\forall a \in A \; aRa$$

Examples

- $\leq$, $\geq$
- $=$
- $\subseteq$
- $a \equiv b \pmod{n}$

Irreflexivity

We say that a binary relation $R \subseteq A \times A$ is *irreflexive* if

$$\forall x \in A \ (x, x) \notin R$$

Examples:

- $<, >$
- $\subset$
- $\neq$

Transitivity

We say that a relation $R \subseteq A \times A$ is *transitive* if

$$\forall a, b, c \in A \; ((aRb \wedge bRc) \to aRc)$$

Examples:

- $\leq, \geq$
- $<, >$
- $=$
- $\subseteq, \subset$
- $a \equiv b \pmod{n}$

Equivalence relations

An *equivalence relation* is a binary relation that is:

- symmetric
- reflexive
- transitive

Examples:

- $=$
- $a \equiv b \pmod{n}$
- $A \times A$

Equivalence relations

An equivalence relation $R \subseteq A \times A$ separates a set into *equivalence classes*, which are subsets of A that are all related by the relation.

- the relation $=$ on $\mathbb{Z}$ separates $\mathbb{Z}$ into an infinite number of equivalence classes, each of which has only one member
- the equivalence relation given by $a \equiv b \pmod 2$, gives two equivalence classes, the even and odd numbers
- the relation $A \times A$ has one equivalence class that contains all of A

Partial orderings

A *partial ordering* on a set A is a binary relation over A which is:

- reflexive
- transitive
- anti-symmetric

Examples:

- $\leq$, $\geq$
- $\subseteq$

Partial orderings capture the idea of one thing being "before" another in some sense.
If the relation is irreflexive instead of reflexive, it is called a *strict partial ordering*.

Total ordering

A *total ordering* on A is a partial ordering R over A that also has the property:

$$\forall x, y \in A \; (xRy \vee yRx)$$

This means that we can always compare any two elements of A.

Examples:

- $\leq$, $\geq$
- lexicographical ordering

If the relation is irreflexive instead of reflexive, it is called a *strict total ordering*.

Example: ancestry

Let H be the set of all humans. Define R over H by aRb if b is an ancestor of a. I.e. b is a parent, grandparent, great-grandparent, etc. of a.
Is R:

- Symmetric?
- Anti-symmetric?
- Transitive?
- Reflexive?
- Irreflexive?
- An equivalence relation, (strict) partial ordering or total ordering?

Example: marriage

Let H be the set of all humans in Australia. Define R over H by aRb if a is married to b.
Is R:

- Symmetric?
- Anti-symmetric?
- Transitive?
- Reflexive?
- Irreflexive?
- An equivalence relation, (strict) partial ordering or total ordering?
- What if you also say aRa for all a?

Example: city location

Let S be the set of all cities in Australia. Define R over S by aRb if a is south of b or at the same latitude as b.
Is R:

- Symmetric?
- Anti-symmetric?
- Transitive?
- Reflexive?
- Irreflexive?
- An equivalence relation, (strict) partial ordering or total ordering?

Hashes

Let S be the set $\{0,1\}^*$ of bit strings of any length and let $H(x)$ be the SHA256 hash* of x. Define R over S by aRb if $H(a) = H(b)$.

Is R:

- Symmetric?
- Anti-symmetric?
- Transitive?
- Reflexive?
- Irreflexive?
- An equivalence relation, (strict) partial ordering or total ordering?

* The cyrptographic hash function SHA256 is an algorithm that maps data of arbitrary size to a string of 256 bits.

Relations in Python

- Python has several built in relations, eg.
  `==, !=, >=, <=, <, >`
- Internally, these are all functions that take two arguments and return a `bool`. They are called *operators* in Python, just like `-, /, *` etc.
- The relations all have a function form, eg.
  ```
  operator.eq(a,b)
  ```
  does exactly the same thing as
  ```
  a == b
  ```
- Python doesn't support adding new relations with the *aRb* syntax (infix notation) but workarounds exist
- To make a custom relation, define a function that takes two arguments and returns True or False
- For a custom class, you can define relations (and operators) using existing symbols

Example Python relation

```
def equivMod7(a, b):    # custom relation is just a function
   return (a - b) % 7 == 0

class myClass():
   def __init__(self, x):
      self.x = x
   def __gt__(self, y): # overload the > operator
      return False      # trivial (empty) relation

>>> x = myClass(3)
>>> x > 4               # calling the overloaded > operator
False                   # Always returns False!
>>> x > 0
False                   # Always returns False!
>>>
```

Outline

Functions

A *function* is a relation f between A and B where for each $a \in A$ there is exactly one $b \in B$ such that $(a, b) \in f$. In other words:

$$((a, b) \in f \wedge (a, c) \in f) \rightarrow b = c$$

We write:

$$f : A \rightarrow B$$

and $f(a)$ is the unique $b \in B$ such that $(a, b) \in f$.

Functions

We usually demand that a function $f : A \to B$ is defined an *all* of A. I.e.

$$\forall a \in A\ \exists b \in B\ (a, b) \in f$$

We can use an extra bit of notation ! meaning *unique* to give a concise definition of when a relation f is a function:

$$\forall a \in A\ \exists! b \in B\ (a, b) \in f$$

Domain and range

For a function $f : A \to B$, A is called the *domain* of f and B is called the *co-domain*. The set

$$\{f(x) : x \in A\}$$

is called the *range* of f.

The domain is the set of all elements for which f is defined, that is the possible "inputs" to f. The range is the set of all possible "outputs" from the function.

Domain problems?

Consider the function $f(x) = 1/x$. f is not defined for $x = 0$, but we still call it a function with domain $\mathbb{R} \setminus \{0\}$.

$$f : \mathbb{R} \setminus \{0\} \to \mathbb{R}$$

Function examples

Some functions:

- $\{(1, \pi), (2, \beta), (3, \delta), (4, \gamma)\}$
- $\{(1, \pi), (2, \pi), (3, \pi), (4, \pi)\}$
- $\{(a, b) \in \mathbb{R}^2 : b = a^2\}$
- $\{(a, b) \in \mathbb{R}^2 : b = 2a + 6\}$
- $= (S \to S$ for any $S)$
- (x, y)co-ordinates for drawing a 45 degree line (or horizontal line)

(for appropriate domains and co-domains)

Non-math function examples

- $f(x)$ given by the last name of the QUT student with student number x.
- $f(x)$ given by the URL for the first response for a google search on x
- $f(x)$ given by the MD5 hash of the bit string x
- $f(x)$ given by the string “w00t” for any input x

Non-function examples

These are not functions:

- $\{(1, \pi), (1, \gamma)\}$
- $\{(a, b) \in \mathbb{R}^2 : a = b^2\}$
- $\leq$, $\geq$
- (x, y) co-ordinates for drawing a happy face

Composing functions

Suppose we have $f : A \to B$ and $g : B \to C$. Then we can define

$$g \circ f : A \to C$$

given by

$$(g \circ f)(x) = g(f(x))$$

called g *of* f *of* x.
Formally:

$$g \circ f = \{(x, z) \in A \times C : \exists y \in B \ \ (x, y) \in f \wedge (y, z) \in g\}$$

Inverses

Some functions have a partner, called its *inverse*. Given $f : A \to B$, the inverse $f^{-1} : B \to A$ is a function such that

$$\forall x \in A \ (f^{-1} \circ f(x) = x).$$

Note that the range of f must match the domain of f^{-1}, and the range of f^{-1} must be the domain of f.

Not all functions have inverses. Example: $f(x) = 0$ has no inverse.

Functions in Python

Python has its own notion of what a function is, and it isn't the same!

- Every computable mathematical function can be written as a function in Python
- Functions in Python that are *side-effect free* and *deterministic* are also functions in the mathematical sense
- Side-effect free means that the function doesn't modify the state of the program (including I/O)
- Deterministic means that there is no randomness
- Calling such a function twice is the same as calling it once

Function examples in Python

```
def myPolynomial(x):       # function in the mathematical sense
   return x ** 2 + 3 * x + 1

x = 3
def changex(y):            # modifies state, not a function
   global x                # in the mathematical sense
   x = y

import random as R
def d6():                  # not deterministic, not a function
   return R.randint(1,6)   # in the mathematical sense
```

Partial functions

Sometimes we don't care about the entire domain:

- Only have data about some particular items
- Using a large set to encode a small number of items (eg. set of numbers to represent student numbers)

In these cases we might define a *partial function*

Definition

A *partial function* f from a set S to a set T is a function from a subset of S to T.

In terms of relations, partial function f contains *at most* one pair (s, t) for every $s \in S$ compared to a function which has *exactly* one such pair.

Dictionaries as partial functions

In Python we can use dictionaries to implement partial functions.

```
>>> d = { 'one': 1, 'tree': 4, 'bark': 't' } # dictionary literal
>>> d['one']                                 # access item
1
>>> d[17] = 'seven'                          # set item
>>> d
{'one': 1, 'tree': 4, 'bark': 't', 17: 'seven'}
>>> d.keys()                                 # get the domain
dict_keys(['one', 'tree', 'bark', 17])
>>> d.values()                               # get the range
dict_values([1, 4, 't', 'seven'])
>>> d[17] = 'seventeen'                      # update item
>>> d
{'one': 1, 'tree': 4, 'bark': 't', 17: 'seventeen'}
```

Outline

Relations

Functions

Sequences

Sequences

Sometimes we care about some set of items, but the order matters:

- Bytes in a file
- Ranked sports teams in a league
- Sorted list used for binary search

We use *sequences* for this job. These are often implemented as *lists* or *arrays*.

Formal definition

Definition

A *sequence* of length n is a function $x : \{1 \ldots n\} \to S$ for some set S.

Sequences are often notated like x_j rather than $x(j)$. Sequences are also often written like

$$[1, 5, 7, 1]$$

but this is not universal.

Sequences is Python: lists

```
>>> l = [ 1, 5, 6 ]          # A list literal
>>> l[2] = 17                # set list item 2
>>> l[2]                     # access list item 2
17
>>> l[0]                     # indices start from 0
1
>>> l.append(7)              # add new item to end
>>> l
[1, 5, 17, 7]
>>> l.pop()                  # remove and return last item
7
>>> len(l)                   # length of list
3
>>> 17 in l                  # item in the list?
True
```

Sequences in Python: tuples

```
>>> t = (1, 2)     # basic tuple notation
>>> s = (1,)       # tuple with 1 element
>>> s
(1,)
>>> u = t + (3, ) # concatenating tuples
>>> u
(1, 2, 3)
>>> u[2]           # access item in tuple
3
>>> a,b,c = u      # tuple unpacking (also for lists)
>>> print(a,b,c)
1 2 3
```

Tuples vs sequences

Tuples and sequences are very similar!

- Basically two ways of expressing the same thing
- Tuples usually used for, short, fixed length
- In tuples, each position often has a different meaning

In Python:

- Lists are mutable, tuples are immutable (more on this later)
- Generally, use lists for variable length data
- Generally, use tuples if the entries are related (eg. x-y coordinates)

Python containers so far

We've see several containers in Python:

- Tuples
- Sets
- Dictionaries
- Lists

Mutability

In Python some objects can change, others are static

- *mutable* types can change over time: lists, dictionaries, sets
- *immutable* types don't change: tuples, frozensets, numbers, strings
- operations on immutable types return a new object
- operations on mutable types change the object

Mutable examples

```
>>> x = 17
>>> x = x + 1          # object 17 is replaced by new object 18
>>> S = { 1, 2 }
>>> T = S              # S and T refer to the same object
>>> S.add(3)           # set is updated in place
>>> T                  # T reflects change to S
{1, 2, 3}
>>> FS = frozenset({1, 2})
>>> FT = FS            # FT and FS refer to same object
>>> FS = FS | {3}      # FS now refers to a new object
>>> FT                 # FT still refers to old object
frozenset({1, 2})
```

Hashability

Hashable types can be identified by a hash value, used internally by some containers

- mutable types are not hashable
- immutable types are generally hashable
- immutable containers are hashable only if they contain only hashable elements

Set elements and dictionary keys must be hashable.

Hashability exmaples

```
>>> S = { {1}, {2} }  # attempt unhashable in set
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>> S = { frozenset({1}), frozenset({2})}
>>> S
{frozenset({2}), frozenset({1})}
>>> D = { [1]: 'one' }  # unhashable key
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> D = { (1,): 'one' }
```