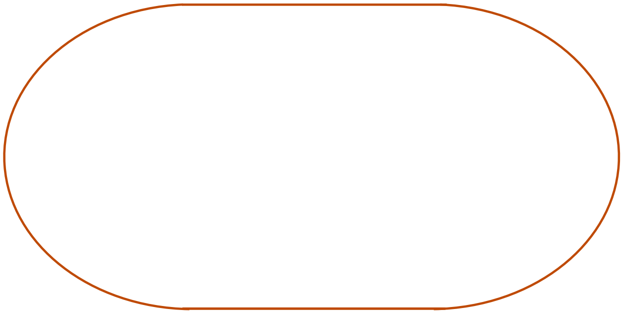
A Gentle Introduction to Metabolic Modelling with Python

Mark Poolman

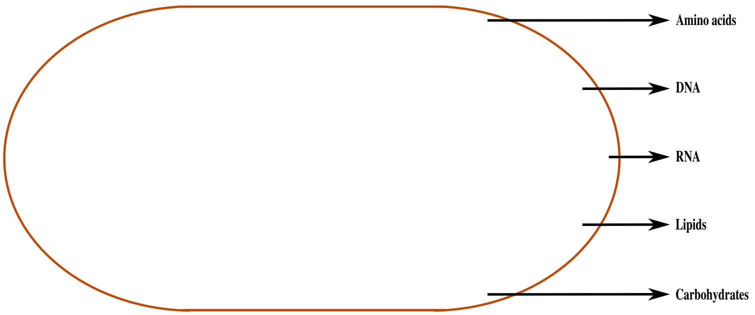
Oxford Brookes University

December 9, 2024

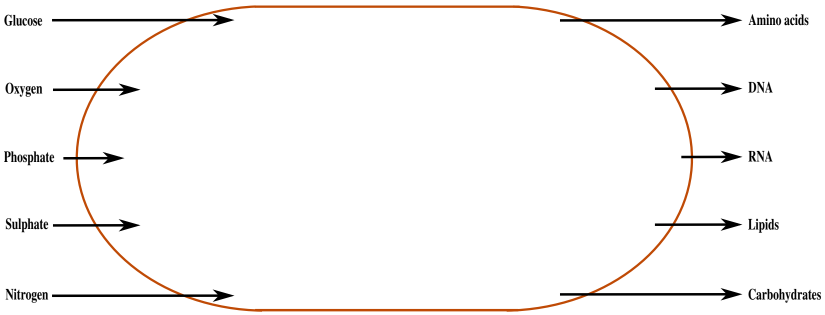
The Problem



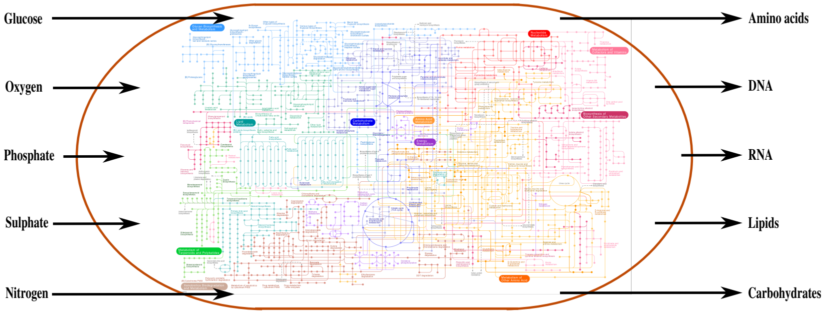
The Problem



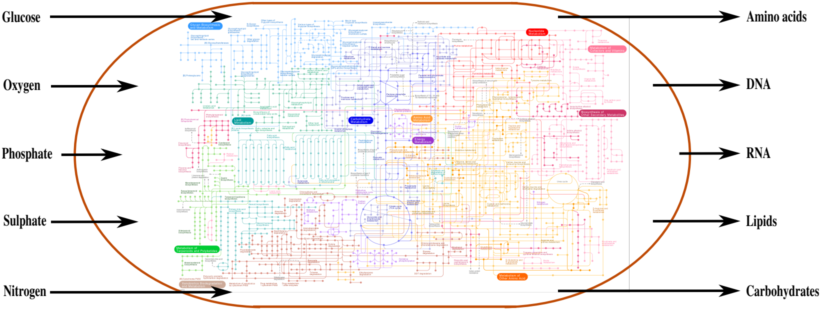
The Problem


Glucose
Oxygen
Phosphate
Sulphate
Nitrogen
Amino acids
DNA
RNA
Lipids
Carbohydrates

The Problem


Glucose
Oxygen
Phosphate
Sulphate
Nitrogen
Amino acids
DNA
RNA
Lipids
Carbohydrates

The Problem



How to connect input(s) to output(s) ??

Motivation

What do we want to know - can we:

Define network behaviour (assign fluxes to reactions)?

Determine the effect of network modification?

Identify the modification needed to achieve a specific effect?

Challenges with large networks

They are large (!)

- Can we extract simple subsystems from very large reaction networks ?
- How do the ‘standard’ biochemical pathways function in very large networks ?
- How will this help our practical understanding of biochemical networks ?

This is *not* a programming course.

This is *not* a programming course.

- No assumption of previous programming experience.
- Basic usage of a language as a tool - no technical details.
- Fundamental mathematical concepts as relevant to network analysis.

Why use a language for modelling ?

- Flexibility - define what you want to do.
- Repeatability - apply the actions same actions to many models.
- Reliability - errors are less likely to go unnoticed, code can be analysed.
- Abstract concepts or large data-sets can't always be visualised.

Why Python ?

- Easy to learn.
- Forgiving.
- Flexible.
- Interactive.
- High level - lets you concentrate on the problem, not the computer.
- Wide range of existing software and libraries.
- Free (As in Beer and Freedom).

What's in a program?

- A collection of *data* representing some real-world entity.
- A set of actions that can be performed on that data.
- Some means by which the user can specify which actions to perform.

In Python (and other languages) the data and actions are both defined by *Objects* (aka *types*).

Objects and types

An object is a computational representation of something that exists in the real world.

The type (or class) of an object is defined by its properties.

- Cats:
  - Fur colour,
  - Length of whiskers.

- Proteins:
  - AA sequence,
  - Iso-electric point.

Objects and types

The type of an object defines **what it can do**, e.g.

- Cats can:
  - Sleep
  - Go miaow
- Proteins can:
  - Precipitate
  - Catalyse a reaction

Objects and types

The type of an object defines **what can be done to it**, e.g.

- Cats can be:
  - Stroked
  - Chased
- Proteins can be:
  - Crystallised
  - Digested

Objects and types

The type of an object defines **how it can interact with other objects**, e.g.

- Cats can:
  - Reproduce with other cats
  - Digest a protein
- Proteins can:
  - Bind to other proteins
  - Poison a cat

Types and Classes - Summary

The concept of objects that have known properties, can be acted upon and can interact with other objects is central.

Objects are abstract representations of their real-world equivalents (including proteins and cats).

(and, of course, metabolic networks)

Types and Classes in Python - Syntax

Attributes define the properties of an object and can either be:

Data attributes MyCat.NumberOfWhiskers

OR

Method attributes Indicated by parentheses ()
MyCat.PlayWithString()

Method attributes can be passed additional information:

MyCat.GotoSleep(3600)

Method attributes can *return* information:

FeedNow = MyCat.IsHungry()

Types and Classes in Python

Python defines a number of built-in fundamental classes, which can be used to create more complex representations of real-world entities.

The distinction between types and classes in Python is historical, in modern python they are the same thing.

Builtin types are subdivided into:

Primitive: Represents exactly one value.

Compound: Can represent multiple values.

Note: Variable types are *not* declared in advance - type is determined by assignment.

Primitive types: Boolean

The simplest of all classes and can take the value of True or False.

e.g. FeedNow = MyCat.IsHungry()

FeedNow is logically a Boolean value: MyCat is either hungry or it is not.

Used (mainly) for various decision making.

Primitive types: Integer

Whole numbers (negative and positive)

Range is only limited by the capacity of the computer:

e.g.

Calculate 10^{10^6}

Massive = 10**10**6

The usual mathematical operators $+$, $-$, $*$, $/$ work *mainly* as expected, but see later.

Primitive types: Floats

Real numbers with possible with a fractional part. Defined either by a decimal and/or 'e' notation

e.g.:

NearPi = 3.12

Planck = 6.62607015e–34

Range is double precision:

$10^x : -308 \leq x \leq 308$

(But only 16 SF)

Standard operators act as before

Python String Class

Strings are sequences of characters, often used for names and simple descriptions, but could also represent an entire document.

- Create an object called text of type string:

```
>>> text = "My cat plays with string"
```

- It has properties, e.g. length:

```
>>> len(text)
24
```

- It can be acted upon, e.g. printed:

```
>>> print (text)
My cat plays with string
```

- It can interact with other objects:

```
print (text + " and mice")
My cat plays with string and mice
```

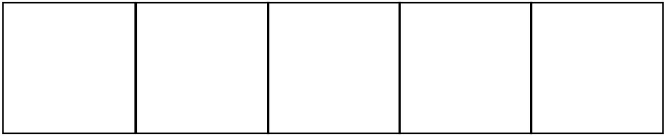
Compound types

Compound types allow arbitrary collections of objects to be held together. The two major compound types are:

- **Lists**: Items are stored in order and are referenced (*indexed*) by an integer.
- **Dictionaries:** Items have no implicit order and can be indexed by a variety of types (commonly strings)

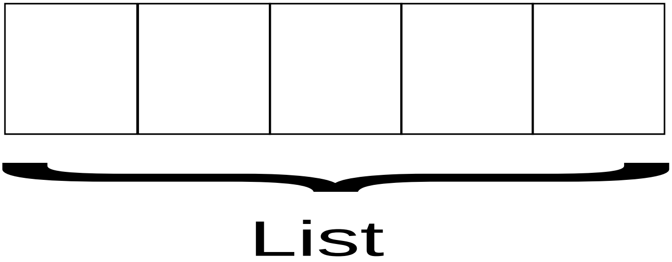
Compound types - Lists

Lists hold collections of *items* in order:



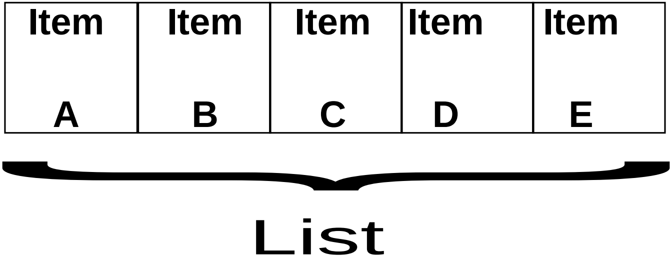
Compound types - lists

Lists hold collections of *items* in order:



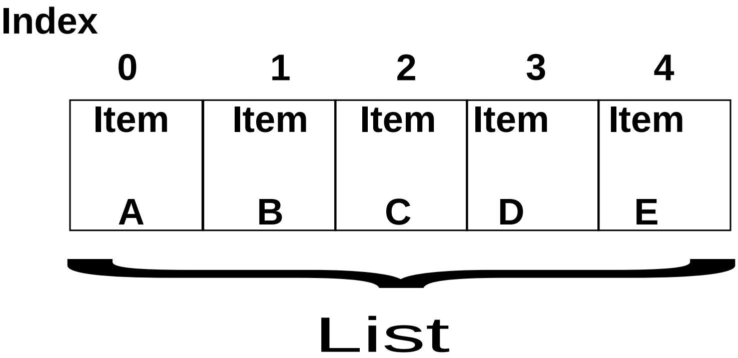
Compound types - lists

Lists hold collections of *items* in order:



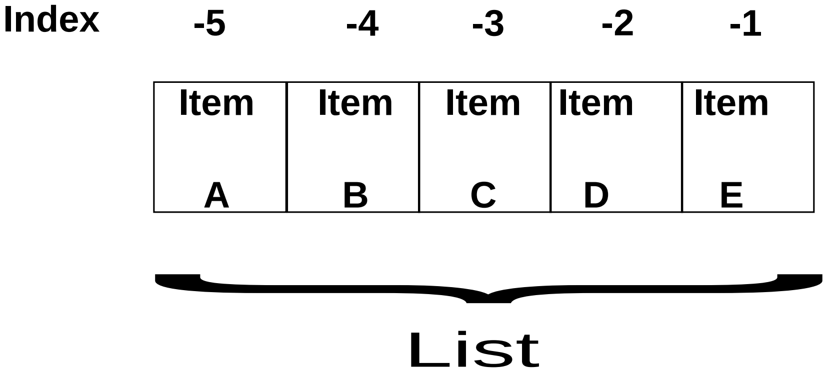
Compound types - lists

Lists hold collections of *items* in order:



Compound types - lists

Lists hold collections of *items* in order:



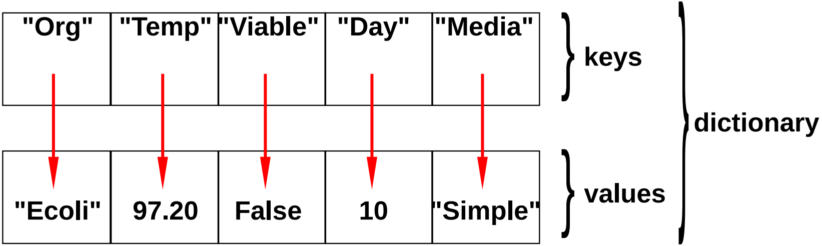
Compound types - lists

Example:

```
>>> ExampleList = ["A","B","C","D","E"]
>>> ExampleList[0]
'A'
>>> ExampleList[1]
'B'
>>> ExampleList[4]
'E'
>>> ExampleList[-1]
'E'
>>> ExampleList[-5]
'A'
```

Compound types - dictionaries

Similar in concept to lists, but items held as *key/value* pairs, are not ordered, and key types are not restricted to integer.



Compound types - dictionaries

Creating a dictionary:

```
>>> ExampleDict={"Org":"Ecoli",
     "Temp":97.2,
     "Viable":False,
     "Day":10,
     "Media":"Simple"
}
```

Changing existing values in a dictionary:

```
>>> ExampleDict["Media"] = "Complex"
>>> ExampleDict["Temp"] = 30
>>> ExampleDict["Viable"] = True
>>> print (ExampleDict)
{'Media': 'Complex',
 'Org': 'Ecoli',
 'Viable': True,
 'Temp': 30,
 'Day': 10
 }
```

Compound types - dictionaries

Adding new key/value pairs to a dictionary:

```
>>> ExampleDict["Recorded by"] = "Mark"
>>> print ExampleDict
{'Media': 'Complex',
 'Org': 'Ecoli',
 'Viable': True,
 'Temp': 30,
 'Recorded by': 'Mark',
 'Day': 10
 }
```

Functions in Python

Functions behave in the same way as class methods, although they are not an attribute of any particular class.

- **dir()** list the *attributes* of an object.
- **type()** returns the class of an object.
- **len()** returns the length of an object (if that is meaningful)

Functions in Python - Examples

```
>>> L = [1,2,3,4]

>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr_
.
.
'append', 'count', 'extend', 'index', 'insert', 'po
'remove', 'reverse', 'sort']

>>> type(L)
<type 'list'>

>>> len(L)
4
>>>
```

Here's one I made earlier - Modules

Modules are used to store pre-written python code for later re-use. They must be *imported* in order to be used:

```
>>> import math
>>> dir(math)
[...,
'pi',...
'sqrt'...]
```

Modules can then be accessed with dot notation:

```
>>> print (math.pi)
3.14159265359
>>> print (math.sqrt(2))
1.41421356237
```

Here's one I made earlier - Modules

Alternatively selection of items can be imported instead:

```
>>> from math import pi, sin
>>> print (sin(pi/4))
0.707106781187
```

For loops (other loops are available)

We frequently wish to act upon each item in a list in turn. The for loop provides a convenient way of doing this.

In general:

```
for Item in MyList :
    # do something
```

Example:

```
>>> for letter in ExampleList:
    print letter
C
B
A
E
D
```

For loops (other loops are available)

For loops provide a convenient way of scanning across a range of numbers, using, for example the built in range function:

```
>>> for x in range(10):
    print (x, x**2, x**3)
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
```