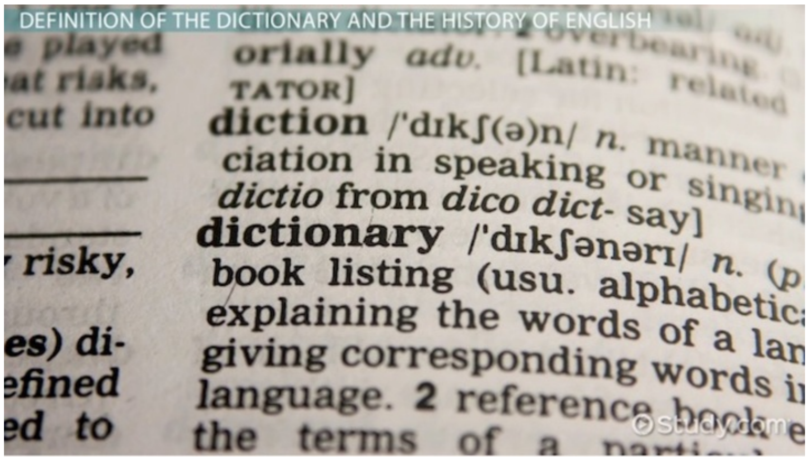
Collect and save data with dict and json
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm
Room A202 Langley Hall

https://people.sc.fsu.edu/∼jburkardt/classes/python_2025/dict/dict.pdf



"The dict data structure"

- *Array storage lets us lookup data by a numeric index;*
- *A* `dict` *is a generalization, which allows us to use a keyword to store and lookup data;*
- *Often this is a much more efficient way to organize information;*
- *Text information is especially well handled this way;*
- *Examples include word dictionaries, information about the states; lyrics organized by song title*
- *Mathematical graphs also work well with dicts;*
- *Even when the data can be regarded as having a numeric index, a dict may offer advantages.*
- *Investigating the Collatz sequence, a dict can be used much more efficiently than an array.*
- *The* `json` *library can save data to a file for later retrieval.*

1 Dictionaries versus indexed lists

Python includes a data type, called a `dict`, which is short for "dictionary". A `dict` can be thought of as a set of values that can be retrieved by keys, instead of a numeric index. This is exactly the way a regular dictionary is used, where we think of a word, and we grab the dictionary, find the word, and expect to find helpful information asssociated with it. Note that the word itself is the key or index that we use to find the corresponding information.

Imagine trying to create a dictionary if we only knew how to work with arrays, that is, data structures in which the key is a numeric index. An English language dictionary would have to be insanely long, having an entry for every possible combination of letters of lengths 1 through 1,185 letters (for a word describing the tobacco mosaic virus). The first 26 entries would be the definitions, or empty spaces, for the words (and non-words!) of one letter. Then we would have 676 more entries, for words and non-words of two letters, and so on. To look up the word "cat", you would need to compute its index in the list, which is something like

```
index = 26 + 676 + (26*26*2 + 0 + 20) - 1
definition = dictionary[index]
```

Of course, because most letter strings are not words, most of the space in the dictionary would be completely wasted.

2 `dict`: A data dictionary

At one time, school children were required to learn the name of the capital of every US state. Supposing we want to practice our knowledge, we would like to be challenged with a random selection of states. How would we go about designing a program that would "know" the state capitals, and be able to respond to any given state name by reporting the corresponding capital?

Knowing how to use arrays, we could simply make two lists, `state` and `capital`, beginning

```
index   state        capital
-----   -------      -----------
    0 ['Alabama',  ['Montgomery',
    1  'Alaska',    'Juneau',
    2  'Arizona',   'Phoenix',
    3  'Arkansas',  'Little Rock',
  ...   ...          ...
   49  'Wyoming']   'Cheyenne']
```

so that our quiz would be something like:

```
  index = np.random.randint ( low = 0, high = 50 )
  print ( 'What is the capital of' + state[index] + '?' )
  guess = input ( response )
  print ( 'Correct answer "' + capital[index] + '".' )
  print ( 'Your answer is "' + capital[index] + '".' )
```

But instead of having to request the capital of Arkansas with the expression `capital[3]`, it seems more natural to say this as `capital['Arkansas']`. The Python `dict` data structure allows us to work in this more natural way.

In this setting, the state is the *key* and the capital is the corresponding `value`. To set up a dict for this information, one way is to define a variable `state_dict` which includes all the pairs of keys and values.

```
state_dict = { \
    'Alabama'  : 'Montgomery',
    'Alaska'   : 'Juneau',
    'Arizona'  : 'Phoenix',
    'Arkansas' : 'Little Rock',
     ...       : ...,
    'Wyoming'    'Cheyenne'
    }
```

As an alternative, we might start with an empty `dict` using a pair of curly brackets (not square brackets!): and then add information as we learn it or discover it:

```
state_dict = {}
state_dict["Pennsylvania"] = "Harrisburg"
state_dict["Virginia"] = "Richmond"
```

No matter how we set the `dict` up, we can then very conveniently access any state capital by using the name of the state as the index. So, if we wish to review every state in our dictionary, we can write

```
 for state, capital in state_dict:
   print ( 'The capital of ' + state + ' is ' + capital + '.' _
```

But for our quiz program, we can use the `.keys()` method to create a list of the states, and then pick a random sample.

```
state = np.random.choice ( list ( state_dict.keys ( ) ) )
print ( 'What is the capital of ' + state + '?' )
print ( 'Correct answer is ' + dict[state] + '.' )
```

There is a similar `.values()` method which can be used to create a list of all the state capitals:

```
capitals = list ( state_dict.values ( ) )
print ( 'Here come all the capitals!' )
print ( capitals )
```

3 The Collatz transformation

One reason for being interested in the Python `dict` data structure is to deal with an issue that we encountered in the previous class, when looking at the Collatz transformation.

In the Collatz transformation, any positive integer n is mapped to a new value $T(n)$, according to the following rule:

```
  if ( n == 1 ):
    Tn = 1
  elif ( ( n % 2 ) == 0 ):
    Tn = n // 2
  else:
    Tn = 3 * n + 1
```

As an example, the Collatz sequence starting at 6 is:

```
6 3 10 5 16 8 4 2 1
```

and the Collatz sequence starting at 19 is

```
19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Notice that the first sequence jumps from 3 to 10, the second from 20 to 10, and after that the sequence are (and must be) identical.

If we are going to compute many Collatz sequences, we can take advantage of this merging process. When we start computing a new sequence, we can check to see if we encounter a value already computed by another sequence. Since that sequence presumaby terminated at 1, so does this one, and we can stop.

4 Using a `dict` to save previous results

We plan to explore the Collatz transformation for many starting values, and we want to take advantage of the fact that if our sequence eventually joins another sequence we already computed, we can stop immediately. But to make this happen, we have to keep some kind of record of the sequences we have already computed.

We can do this by creating a Python `dict`, perhaps called `collatz_dict`. This data structure starts out empty. Given any n, we check to see if it is already a member of the `dict`, that is, already is stored as a key. We do this by testing the logical expression

```
n in collatz_dict
```

If the `dict` already knows about n, so, we can stop. Otherwise, we compute $T(n)$, add $(n, T(n))$ to our information, and then let $T(n)$ become our next value of n.

```
  if ( not collatz_dict ):  # Initialize the dict
    collatz_dict = {}

  while ( not ( n in collatz_dict ) ):
    Tn = collatz ( n )
    collatz_dict[n] = Tn
    n = Tn
```

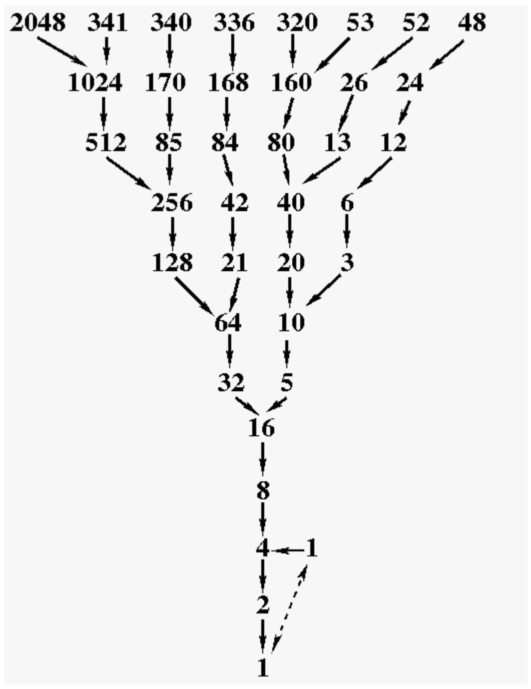
To see the efficiency of this approach, here is what the `dict` looks like after computing Collatz sequences starting at 1, 5 and 15:

```
[(1, 2), (2, 1), (4, 2), (5, 16), (8, 4), (10, 5), (15, 46), (16, 8), (20, 10), (23, 70),
(35, 106), (40, 20), (46, 23), (53, 160), (70, 35), (80, 40), (106, 53), (160, 80)]
```

and here is how the `dict` is updated, after we compute the additional sequences starting at 2, 3, 4, 6, 7, 8, 9 and 10:

```
[(1, 2), (2, 1), (3, 10), (4, 2), (5, 16), (6, 3), (7, 22), (8, 4), (9, 28), (10, 5),
(11, 34), (13, 40), (14, 7), (15, 46), (16, 8), (17, 52), (20, 10), (22, 11), (23, 70),
(26, 13), (28, 14), (34, 17), (35, 106), (40, 20), (46, 23), (52, 26), (53, 160), (70, 35),
(80, 40), (106, 53), (160, 80)]
```

If you continue this experiment by computing the sequence that starts with $n = 27$, you will see that the `dict` is able to compactly incorporate new data for a long sequence whose values range into the thousands. If we had, instead, tried to use an array in some way, or a list, we would have wasted a lot of unused storage, and had to deal with slower access to a much larger data structure.



5 Saving a `dict` with `JSON`

During a Python session, we create variables and fill them with values. When the session is over, all that information is liable to be lost. Sometimes it's enough to print a few values of interest. But what do we do if we want to save some complicated item to be reused later? In particular, once we have created our dictionary structure recording Collatz information, can we store it conveniently before terminating our program, in such a way that we can recover all that information when running a program later?

There are a number of ways of doing this, including the somewhat peculiar Python `pickle` format. However, there is a standard known as `JSON`, whose name stands for JavaScript Object Notation, which is widely used among many different programing languages. Since data stored by `JSON` can be transferred across computer systems and programming languages, we will prefer to use this method for saving and transmitting data.

Information about `JSON` is available at
`https://www.json.org/json-en.html`

We will look at a very simple case, in which our data is already stored in a single variable, a `dict` called `collatz_dict`. To save this information into a file before our program terminates, we do the following:

```
import json
filename = 'collatz_dict.json'
output = open ( filename, 'w' )
json.dump ( collatz.dict, output )
output.close ( )
```

Then, later, we can start up another Python program, read the file, and convert it back into a Python `dict`, to which we can add more information:

```
  filename = 'collatz_dict.json'
  input = open ( filename, 'r' )
  collatz_dict = json.load ( input )
  input.close ( )
```

Using this approach, we were able to save our Collatz computation, and then later recover the information, run Collatz on new starting points, and update our Collatz `dict` efficiently.

6 The JSON file format

When data is saved by `JSON`, the resulting file is a text file that can be typed out, edited, printed, or emailed, or copied to an external device. If you take a look, you will see that the file looks very similar to a Python `dict` definition; it consists of pairs of names and values.

Although we have only considered saving a Python `dict`, any legal Python variable, or set of variables, can be saved in this way. One way to save multiple items in a single file is to combine them into a temporary `dict`. When it is time to retreive the data, it is only necessary to "unravel" the information by requesting the value corresponding to the appropriate keyword.

Here is an example of a what a `JSON` file might look like, which stores information about an image.

```
{
  "id": "0001",
  "type": "donut",
  "name": "Cake",
  "image":
    {
      "url": "images/0001.jpg",
      "width": 200,
      "height": 200
    },
  "thumbnail":
    {
      "url": "images/thumbnails/0001.jpg",
      "width": 32,
      "height": 32
    }
}
```

Assuming this file has been loaded by the command

```
my_image = json.load ( 'cake.json' )
```

we could retrieve the `id` field by

```
  id = my_image["id"]
```

Simlarly, we could get the *url* of the image by the two step process:

```
  image = my_image["image"]
  url = image["url"]
```

This example should suggest how the `JSON` format is flexible enough to handle many kinds of data, and stores that data in a text file that can easily be transferred and then "unwrapped" by a program in almost any programming language.