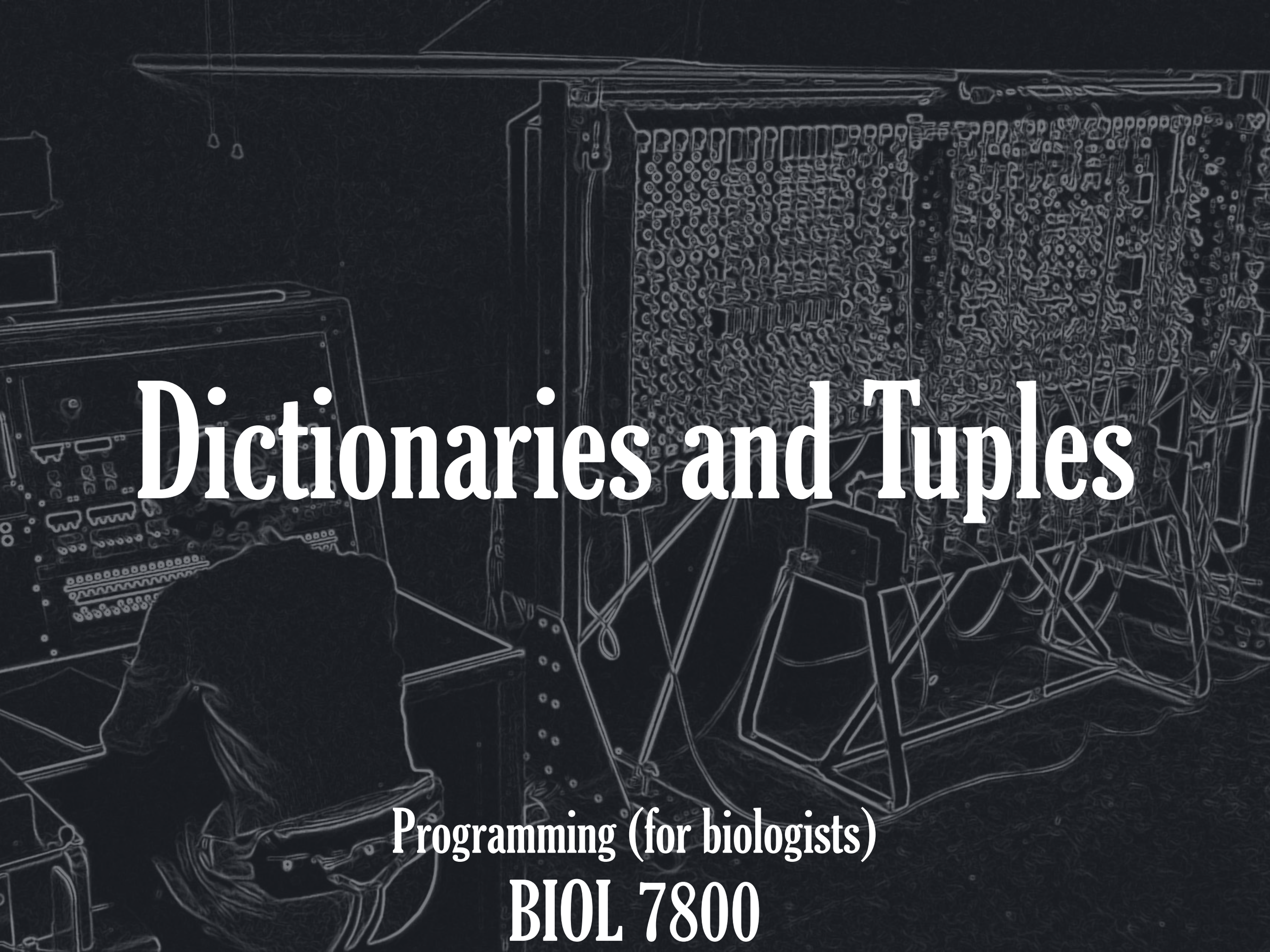


Dictionaries and Tuples

Programming (for biologists)

BIOL 7800

Strings

A **string** is a **sequence** of characters

‘Able was I ere I saw Elba’

These are both “strings”

‘1 2 3 4 5 6 7’

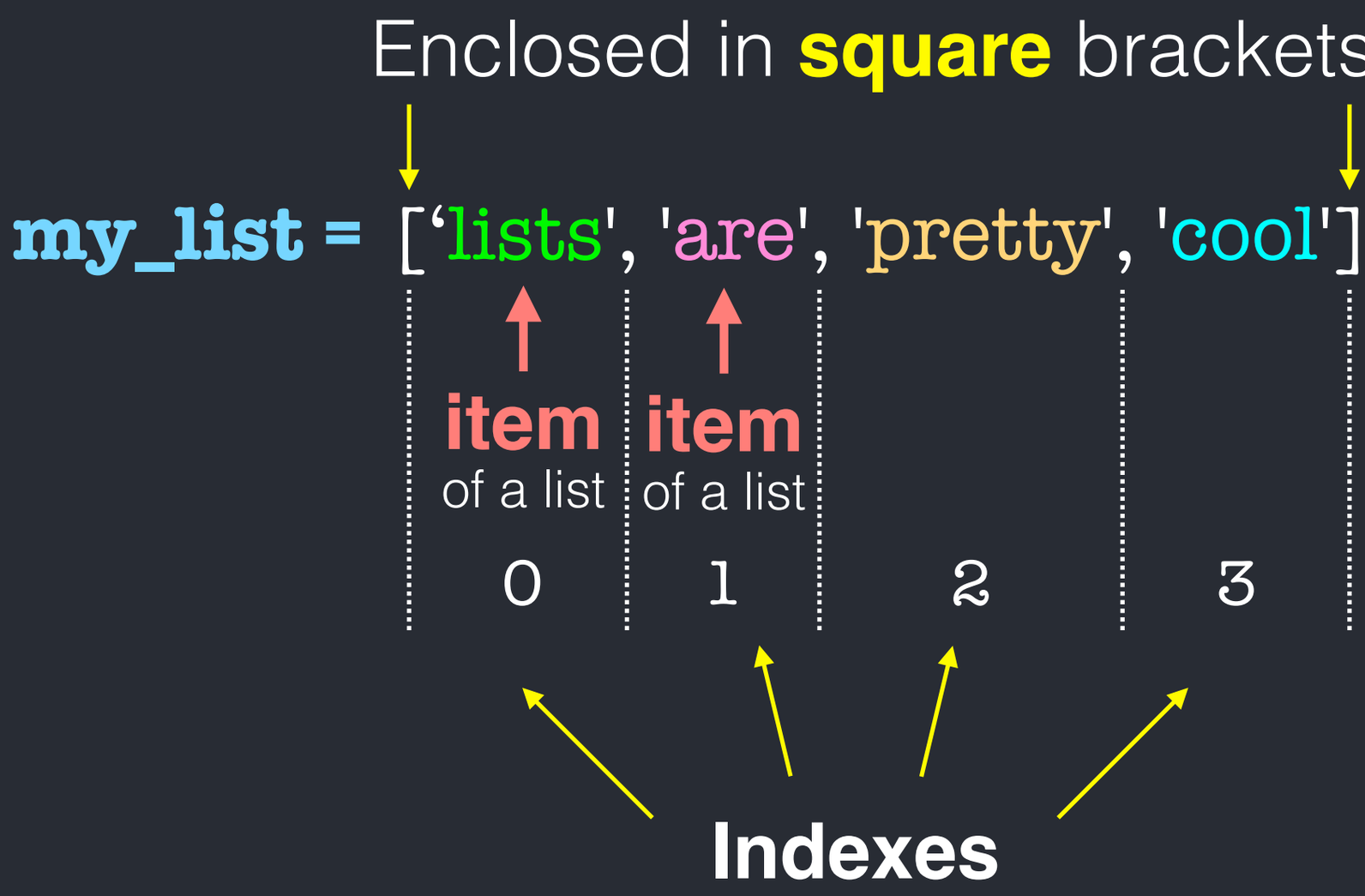
Lists

In: **my_string.split(‘ ’)**
Out: [‘this', 'is', 'my', 'string']

List

Like a string, a **list** is also a sequence of values, but values can be of any **type** (not just characters)

Anatomy of a List



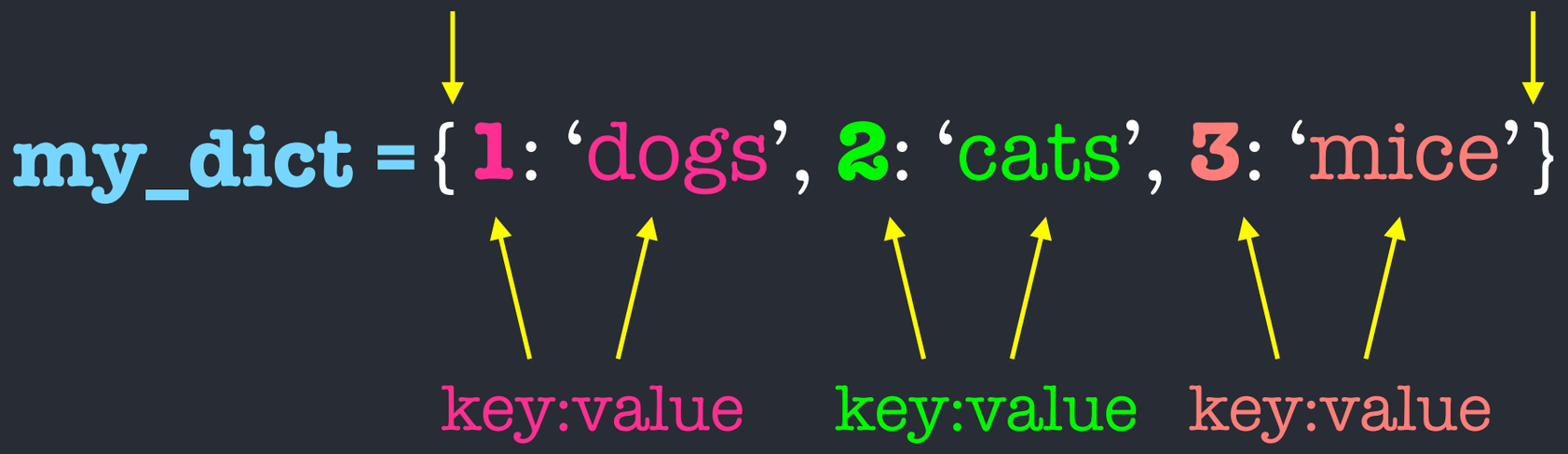
You can access an element

my_list[3] = 'cool'

Anatomy of a Dictionary

A dictionary is a *mapping* of some
value(s) to a unique key

Enclosed in **squiggely** braces



keys **must** be unique!

Anatomy of a Dictionary

A dictionary is a *mapping* of some
value(s) to a unique key

A dictionary allows you to **"look up"**
values that go with keys!

```
my_dict = {1: 'dogs', 2: 'cats', 3: 'mice'}
```

You can access an element
`my_dict[1]` = 'dogs'

But the following does not work:
`my_dict[0]` = 'dogs'

Dicts are generalized lists

```
my_list = ['dicts', 'are', 'pretty', 'cool']
           0        1      2         3
```



In a **list**, the indexes are defined for us by element number

```
my_dict ={0: 'dicts', 1: 'are', 2: 'pretty', 3: 'cool'}
```

In a **dict**, we **define** the indexes

Anatomy of a Dictionary

my_dict = {999: ‘dicts’, 72: ‘are’, 2: ‘pretty’, 63: ‘cool’}

In a **dict**, we **define** the indexes (called “keys”)

(they needn’t have any order)

my_dict = {‘word1’: ‘dicts’, ‘word2’: ‘are’, ‘word3’: ‘pretty’}

In a **dict**, we **define** the keys

(and they don’t need to be numbers)

Keys can be **any unique object**

Creating a Dictionary

```
my_pets = dict()
my_pets = {}
```

Add a key/value

```
In:  my_pets['dogs'] = 2
In:  print(my_pets)
Out: {'dogs': 2}
```

Add several keys/values

```
In:  my_pets = {'dogs': 2, 'cats':3, 'hamsters':1}
In:  print(my_pets)
Out: {'dogs': 2, 'cats':3, 'hamsters':1}
```

Dictionary

```
my_pets = {'dogs': 2, 'cats':3, 'hamsters':1}
```

Access an element

```
In:  my_pets['dogs']
Out: 2
```

Change an element

```
In:  my_pets['dogs'] = 12
In:  my_pets['dogs']
Out: 12
```

Increment an element

```
In:  my_pets['dogs'] += 1
In:  my_pets['dogs']
Out: 13
```

Dictionary

```
my_pets = {'dogs': 2, 'cats':3, 'hamsters':1}
```

Delete an element

```
In:  del my_pets['dogs']
In:  print(my_pets)
Out: {'cats':3, 'hamsters':1}
```

Dictionary

Dictionary keys must be unique
while values *can be anything*

Here, all **values** are the same

```
my_pets = {'dogs': 2, 'cats':2, 'hamsters':2}
```

Here, **values** have **list** type

```
my_pets = {'dogs':['scoobie', 'idiot'],
           'cats':['stinker', 'sir poopsalot']}
```

Here, **values** have **multiple** types

```
my_pets = {'dogs':numpy.array([1,2,3]),
           'cats': 'Seq('ACGTGAGTCGTATA')}
```

Dictionary

Dictionary keys are also **unordered**
Meaning that you **cannot assume** they will:

(1) remain in the order you entered them
(2) follow any logical order

```
In:  test_dict = {'A': 'best', 'B': 'good', 'C': 'okay', 'D': 'bad'}
In:  print(test_dict)
Out: {'B': 'good','D': 'bad', 'C': 'okay', 'A': 'best', }
```

Dictionary

```
In: my_pets = {'dogs':['scoobie', 'idiot'],
               'cats':['stinker', 'sir poopsalot']}
In: len(my_pets)
```

What is the **len** of **my_pets**?

Dictionary

```
In:  my_pets = {'dogs':['scoobie', 'idiot'],
                'cats':['stinker', 'sir poopsalot']}
In:  len(my_pets)
Out: 2
```

len gives the total count of `key:value` pairs in the **dict**

Dictionaries are iterable

```
In:  my_pets = {'dogs': 2, 'cats':2, 'hamsters':2}
In:  for item in my_pets:
         print(item)
dogs
cats
hamsters
```

But, standard iteration only returns their `keys`

Dictionaries are iterable

```
In:  my_pets = {'dogs': 2, 'cats':2, 'hamsters':2}
In:  for item in my_pets:
         print(item, my_pets[item])
dogs 2
cats 2
hamsters 2
```

To get the `value` associated with each `key`
we have to iterate over `keys`, and lookup each `value`

This is **slow**!

Dictionaries are iterable

```
In:  my_pets = {'dogs': 2, 'cats':2, 'hamsters':2}
In:  for key, value in my_pets.items():
         print(key, value)
dogs 2
cats 2
hamsters 2
```

We can use the `.items()` method of dictionaries to iterate over all `key:value` pairs in the dictionary

This is **faster**!

Dictionary methods

Get **only the keys** of a dictionary with .keys()

```
In:  my_pets = {'dogs': 2, 'cats':2, 'hamsters':2}
In:  my_pets.keys()
Out: dict_keys(['dogs', 'cats', 'hamsters'])
```

Get **only the values** of a dictionary with .values()

```
In:  my_pets = {'dogs': 2, 'cats':2, 'hamsters':2}
In:  my_pets.values()
Out: dict_values([2, 2, 2])
```

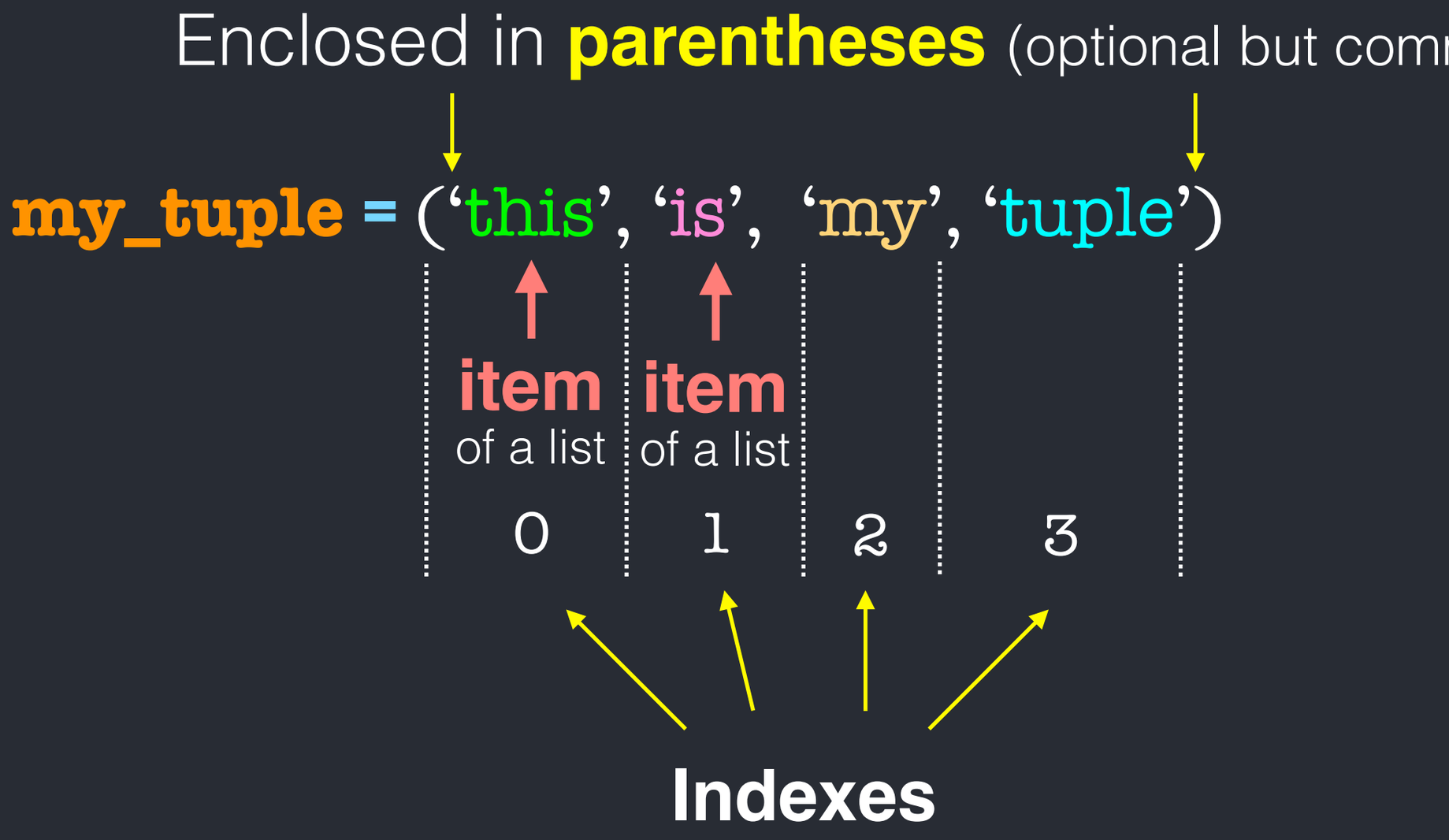
Tuple

In: my_tuple = (‘this’, 'is', 'my', ‘tuple’)

Tuple

Like a list, a **tuple** is a sequence of values.
Values can be of any **type** and values are **immutable**

Anatomy of a Tuple



You can access an element

`my_tuple[1] = ‘is’`

Anatomy of a Tuple

This is **also** a tuple

Comma separated values (no parens)



Creating a Tuple

my_tuple = ()

Set a variable to opposing parentheses

to create empty tuple

my_tuple = **tuple**()

Set a variable to tuple() type

my_tuple = (‘dog’, ‘cat’, ‘mouse’, ‘rat’)

Type in tuple entries between opposing parentheses

my_tuple = tuple(‘dog’)

“tuplify” a string

(‘d', 'o', ‘g')

my_tuple = tuple([‘dog’, ‘cat’, ‘rat’])

“tuplify” a list

Creating a Tuple

This is **not** a tuple

```
my_not_a_tuple = ('a')
```

A single element must be followed by a comma

```
my_tuple = ('a',)
```

or

```
my_tuple ='a',
```

Tuples are immutable

Unlike lists, you **cannot** modify tuple elements

```
In:  my_tuple = ('dog', 'cat', 'mouse', 'rat')
In:  my_tuple[1] = 'hamster'
Out: TypeError: 'tuple' object does not support item assignment
```

Tuples are immutable

Unlike lists, you **cannot** modify list elements

But what about…

```
In:  my_tuple = ([1,2,3], ['cat','dog'], [4.6])
In:  my_tuple[1][0] = 'rabbit'
In:  print(my_tuple)
```

Tuples are immutable

Unlike lists, you **cannot** modify list elements

But what about…

```
In:  my_tuple = ([1,2,3], ['cat','dog'], [4.6])
In:  my_tuple[1][0] = 'rabbit'
In:  print(my_tuple)
Out: ([1,2,3], ['rabbit','dog'], [4.6])
```

How ??

Tuple operations

You can **add** (concatenate) tuples

In: `('a', 'b', 'c') + ('1', '2', '3')`
Out: `('a', 'b', 'c', '1', '2', '3')`

You can **multiply** (repeat) tuples

In: `('a',) * 6`
Out: `('a','a','a','a','a','a')`

But, tuples have many fewer attributes/methods
[so no `.append()` or `.extend()`]

Tuples & Functions

example.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def function1(arg):
6       product = arg * arg
7       return product
8
9
10  def main():
11      result = function1(2)
12      print result
13
14  if __name__ == '__main__':
15      main()
16
```

return product

Typically, functions return one value

File 0 | Project 0 | No Issues | example.py* | 7:5 | LF | UTF-8 | Python

Tuples & Functions

example.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def function1(arg):
6       product = arg * arg
7       return arg, product, arg
8
9
10  def main():
11      result = function1(2)
12      print(result)
13
14
15  if __name__ == '__main__':
16      main()
17
```

return arg, product, arg

But, functions
can return more
than one value
in form of a tuple

So, what does this print?

File 0 | Project 0 | No Issues | example.py | 12:18 | LF | UTF-8 | Python | 1 update

Tuples & Functions

example.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def function1(arg):
6       product = arg * arg
7       return arg, product, arg
8
9
10  def main():
11      result = function1(2)
12      print(result)
13
14
15  if __name__ == '__main__':
16      main()
17
```

return arg, product, arg

But, functions
can return more
than one value
in form of a tuple

(2, 4, 2)

File 0 | Project 0 | No Issues | example.py | 12:18 | LF | UTF-8 | Python | 1 update

Tuples & Functions

example.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def function1(arg):
6       product = arg * arg
7       return arg, product, arg
8
9
10  def main():
11      r_arg1, product, r_arg2 = function1(2)
12      print("This is 1st pos", r_arg1)
13      print("This is 2nd pos", product)
14      print("This is 3rd pos", r_arg2)
15
16
17  if __name__ == '__main__':
18      main()
```

return arg, product, arg

We can "unpack" those three **return** values into 3 variables

File 0 | Project 0 | ✔ No Issues | example.py | 18:11 | LF | UTF-8 | Python | 1 update

zip() and Tuples

zip() is a function that joins two sequences to make one tuple
the resulting tuple has one element from each sequence

```
In:   jackson = 'abc'
In:   five = '123'
In:   print(list(zip(jackson, five)))
Out:  [('a', '1'), ('b', '2'), ('c', '3')]
```

zip() and Tuples

zip() is a function that joins two sequences to make one tuple and can also be used to quickly make a dictionary

```
In:   jackson = 'abc'
In:   five = '123'
In:   my_dict = dict(zip(jackson, five))
In:   print(my_dict)
Out:  {'a': '1', 'b': '2', 'c': '3'}
```