Practical Computing for Scientists

Armin Sobhani

CSCI 2000U

UOIT – Fall 2015

UNIVERSITY OF ONTARIO
INSTITUTE OF TECHNOLOGY

Python
Tuples

by Greg Wilson

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

A *list* is a mutable heterogeneous sequence

A *list* is a mutable heterogeneous sequence

A tuple is an *immutable* heterogeneous sequence

A *list* is a mutable heterogeneous sequence

A tuple is an *immutable* heterogeneous sequence

i.e., a list that can't be changed after creation

A *list* is a mutable heterogeneous sequence

A tuple is an *immutable* heterogeneous sequence

I.e., a list that can't be changed after creation

Why provide a less general type of collection?

A *list* is a mutable heterogeneous sequence

A tuple is an *immutable* heterogeneous sequence

I.e., a list that can't be changed after creation

Why provide a less general type of collection?

Full explanation will have to wait for lecture on sets and dictionaries

A *list* is a mutable heterogeneous sequence

A tuple is an *immutable* heterogeneous sequence

I.e., a list that can't be changed after creation

Why provide a less general type of collection?

Full explanation will have to wait for lecture on sets and dictionaries

Useful even before then

Create tuples using () instead of []

Create tuples using () instead of []

Still index using [] (because everything does)

Create tuples using () instead of []

Still index using [] (because everything does)

```
>>> primes = (2, 3, 5, 7)
>>> print(primes[0], primes[-1])
2 7
>>>
```

Create tuples using () instead of []

Still index using [] (because everything does)

```
>>> primes = (2, 3, 5, 7)
>>> print(primes[0], primes[-1])
2 7
>>> empty_tuple = ()
>>> print(len(empty_tuple))
0
>>>
```

Create tuples using () instead of []

Still index using [] (because everything does)

```
>>> primes = (2, 3, 5, 7)
>>> print(primes[0], primes[-1])
2 7
>>> empty_tuple = ()
>>> print(len(empty_tuple))
0
>>>
```

Must use `(val,)` for tuple with one element

Create tuples using () instead of []

Still index using [] (because everything does)

```
>>> primes = (2, 3, 5, 7)
>>> print(primes[0], primes[-1])
2 7
>>> empty_tuple = ()
>>> print(len(empty_tuple))
0
>>>
```

Must use `(val,)` for tuple with one element

Because math says that (5) is just 5

Create tuples using () instead of []

Still index using [] (because everything does)

```
>>> primes = (2, 3, 5, 7)
>>> print(primes[0], primes[-1])
2 7
>>> empty_tuple = ()
>>> print(len(empty_tuple))
0
>>>
```

Must use `(val,)` for tuple with one element

Because math says that (5) is just 5

One of Python's few syntactic warts...

Don't need parentheses if context is enough

Don't need parentheses if context is enough

```
>>> primes = 2, 3, 5, 7
>>> print(primes)
(2, 3, 5, 7)
>>>
```

Don't need parentheses if context is enough

```
>>> primes = 2, 3, 5, 7
>>> print(primes)
(2, 3, 5, 7)
>>>
```

Can use on the left of assignment

Don't need parentheses if context is enough

```
>>> primes = 2, 3, 5, 7
>>> print(primes)
(2, 3, 5, 7)
>>>
```

Can use on the left of assignment

```
>>> left, middle, right = 2, 3, 5
>>>
```

Don't need parentheses if context is enough

```
>>> primes = 2, 3, 5, 7
>>> print(primes)
(2, 3, 5, 7)
>>>
```

Can use on the left of assignment

```
>>> left, middle, right = 2, 3, 5
>>> print(left)
2
>>> print(middle)
3
>>> print(right)
5
>>>
```

Don't need parentheses if context is enough

```
>>> primes = 2, 3, 5, 7
>>> print(primes)
(2, 3, 5, 7)
>>>
```

Can use on the left of assignment

```
>>> left, middle, right = 2, 3, 5
>>> print(left)
2
>>> print(middle)
3
>>> print(right)
5
>>>
```

With great power comes great responsibility...

Allows functions to return multiple values

Allows functions to return multiple values

```
>>> def bounds(values):
...     low = min(values)
...     high = max(values)
...     return (low, high)
...
>>>
```

Allows functions to return multiple values

```
>>> def bounds(values):
...     low = min(values)
...     high = max(values)
...     return (low, high)
...
>>> print(bounds([3, -5, 9, 4, 17, 0]))
(-5, 17)
>>>
```

Allows functions to return multiple values

```
>>> def bounds(values):
...     low = min(values)
...     high = max(values)
...     return (low, high)
...
>>> print(bounds([3, -5, 9, 4, 17, 0]))
(-5, 17)
>>> least, greatest = bounds([3, -5, 9, 4, 17, 0])
>>> print(least)
5
>>> print(greatest)
17
>>>
```

Sometimes used to return (success, result) pairs

Sometimes used to return (success, result) pairs

```
def read_if_available(datafile_name):
  if file_exists(datafile_name):
    ...
    return (True, data_values)
  else:
    return (False, [])
```

Sometimes used to return (success, result) pairs

```
def read_if_available(datafile_name):
  if file_exists(datafile_name):
    ...
    return (True, data_values)
  else:
    return (False, [])

success, data = read_if_available('mydata.dat')
if success:
  ...
```

Provides a quick way to swap variables' values

Provides a quick way to swap variables' values

```
>>> left, right = 0, 10
>>>
```

Provides a quick way to swap variables' values

```
>>> left, right = 0, 10
>>> right, left = left, right
>>>
```

Provides a quick way to swap variables' values

```
>>> left, right = 0, 10
>>> right, left = left, right
>>> print(right)
0
>>> print(left)
10
>>>
```

Provides a quick way to swap variables' values

```
>>> left, right = 0, 10
>>> right, left = left, right
>>> print(right)
0
>>> print(left)
10
>>>
```

Python creates temporaries if needed

Provides a quick way to swap variables' values

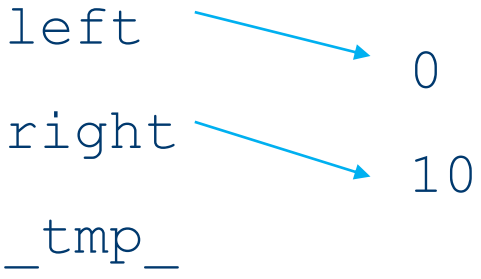
```
>>> left, right = 0, 10
>>> right, left = left, right
>>> print(right)
0
>>> print(left)
10
>>>
```

Python creates temporaries if needed

Provides a quick way to swap variables' values

```
>>> left, right = 0, 10
>>> right, left = left, right
>>> print(right)
0
>>> print(left)
10
>>>
```

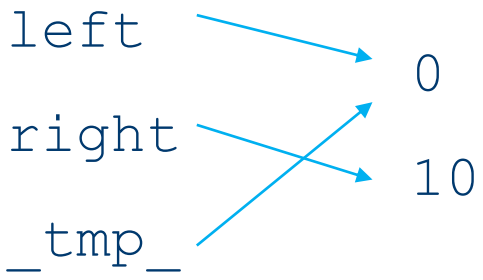
Python creates temporaries if needed



Provides a quick way to swap variables' values

```
>>> left, right = 0, 10
>>> right, left = left, right
>>> print(right)
0
>>> print(left)
10
>>>
```

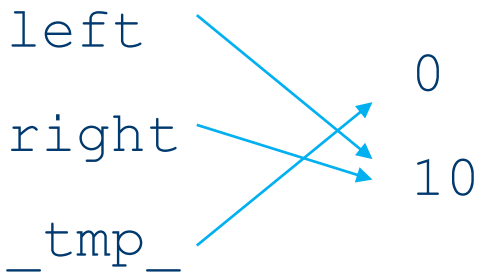
Python creates temporaries if needed



Provides a quick way to swap variables' values

```
>>> left, right = 0, 10
>>> right, left = left, right
>>> print(right)
0
>>> print(left)
10
>>>
```

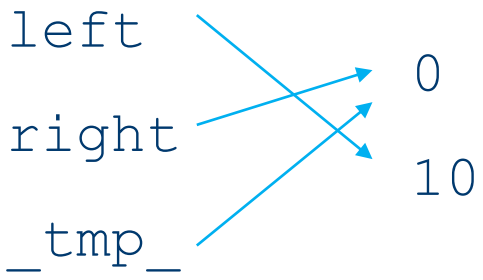
Python creates temporaries if needed



Provides a quick way to swap variables' values

```
>>> left, right = 0, 10
>>> right, left = left, right
>>> print(right)
0
>>> print(left)
10
>>>
```

Python creates temporaries if needed



Provides a quick way to swap variables' values

```
>>> left, right = 0, 10
>>> right, left = left, right
>>> print(right)
0
>>> print(left)
10
>>>
```

Python creates temporaries if needed

And an easy way to unpack a list

And an easy way to unpack a list

```
>>> colors = ['yellow', 'magenta', 'lavender']
>>>
```

And an easy way to unpack a list

```
>>> colors = ['yellow', 'magenta', 'lavender']
>>> left, middle, right = colors
>>>
```

And an easy way to unpack a list

```
>>> colors = ['yellow', 'magenta', 'lavender']
>>> left, middle, right = colors
>>> print(left)
yellow
>>> print(middle)
magenta
>>> print(right)
lavender
>>>
```

And an easy way to unpack a list

```
>>> colors = ['yellow', 'magenta', 'lavender']
>>> left, middle, right = colors
>>> print(left)
yellow
>>> print(middle)
magenta
>>> print(right)
lavender
>>>
```

Number of values must be the same

Often used in loops

Often used in loops

```
>>> pairs = ((1, 10), (2, 20), (3, 30), (4, 40))
>>>
```

Often used in loops

```
>>> pairs = ((1, 10), (2, 20), (3, 30), (4, 40))
>>> for p in pairs:
...  print(p[0] + p[1])
```

Often used in loops

```
>>> pairs = ((1, 10), (2, 20), (3, 30), (4, 40))
>>> for p in pairs:
...     print(p[0] + p[1])
```

Often used in loops

```
>>> pairs = ((1, 10), (2, 20), (3, 30), (4, 40))
>>> for (low, high) in pairs:
...  print(low + high)
```

Often used in loops

```
>>> pairs = ((1, 10), (2, 20), (3, 30), (4, 40))
>>> for (low, high) in pairs:
...  print(low + high)
...
11
22
33
44
>>>
```

The enumerate function produces (index, value) pairs

The enumerate function produces (index, value) pairs

```
>>> colors = ['yellow', 'magenta', 'lavender']
>>> for (i, name) in enumerate(colors):
...  print(i, name)
```

The enumerate function produces (index, value) pairs

```
>>> colors = ['yellow', 'magenta', 'lavender']
>>> for (i, name) in enumerate(colors):
...  print(i, name)
...
0 yellow
1 magenta
2 lavender
>>>
```

The enumerate function produces (index, value) pairs

```
>>> colors = ['yellow', 'magenta', 'lavender']
>>> for (i, name) in enumerate(colors):
...  print(i, name)
...
0 yellow
1 magenta
2 lavender
>>>
```

Prefer this to `range(len(values))`

Assignment-3

- Approximating π

Python

Text

by Greg Wilson

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

How to represent characters?

How to represent characters?

American English in the 1960s:

How to represent characters?

American English in the 1960s:

26 characters × {upper, lower}

How to represent characters?

American English in the 1960s:

26 characters × {upper, lower}

\+ 10 digits

How to represent characters?

American English in the 1960s:

26 characters × {upper, lower}

\+ 10 digits

\+ punctuation

\+

How to represent characters?

American English in the 1960s:

26 characters × {upper, lower}

\+ 10 digits

\+ punctuation

\+ special characters for controlling teletypes

(new line, carriage return, form feed, bell, …)

How to represent characters?

American English in the 1960s:

26 characters × {upper, lower}

\+ 10 digits

\+ punctuation

\+ special characters for controlling teletypes

(new line, carriage return, form feed, bell, …)

= 7 bits per character (ASCII standard)

How to represent text?

How to represent text?

1. Fixed-width records

How to represent text?

1. Fixed-width records

```
A crash reduces
your expensive computer
to a simple stone.
```

How to represent text?

1. Fixed-width records

A crash reduces
your expensive computer
to a simple stone.

A		c	r	a	s	h		r	e	d	u	c	e	s	·	·	·	·	·	·	·	·
y	o	u	r		e	x	p	e	n	s	i	v	e		c	o	m	p	u	t	e	r
t	o		a		s	i	m	p	l	e		s	t	o	n	e	.	·	·	·	·	·

How to represent text?

1. Fixed-width records

A crash reduces
your expensive computer
to a simple stone.

A		c	r	a	s	h		r	e	d	u	c	e	s	·	·	·	·	·	·	·	·
y	o	u	r		e	x	p	e	n	s	i	v	e		c	o	m	p	u	t	e	r
t	o		a		s	i	m	p	l	e		s	t	o	n	e	.	·	·	·	·	·

Easy to get to line N

How to represent text?

1. Fixed-width records

A crash reduces
your expensive computer
to a simple stone.

A		c	r	a	s	h		r	e	d	u	c	e	s	·	·	·	·	·	·	·	·
y	o	u	r		e	x	p	e	n	s	i	v	e		c	o	m	p	u	t	e	r
t	o		a		s	i	m	p	l	e		s	t	o	n	e	.	·	·	·	·	·

Easy to get to line N

But may waste space

How to represent text?

1. Fixed-width records

```
A crash reduces
your expensive computer
to a simple stone.
```

A		c	r	a	s	h		r	e	d	u	c	e	s	·	·	·	·	·	·	·	·
y	o	u	r		e	x	p	e	n	s	i	v	e		c	o	m	p	u	t	e	r
t	o		a		s	i	m	p	l	e		s	t	o	n	e	.	·	·	·	·	·

Easy to get to line N

But may waste space

What if lines are longer than the record length?

How to represent text?

1. Fixed-width records
2. Stream with embedded end-of-line markers

How to represent text?

1. Fixed-width records
2. Stream with embedded end-of-line markers

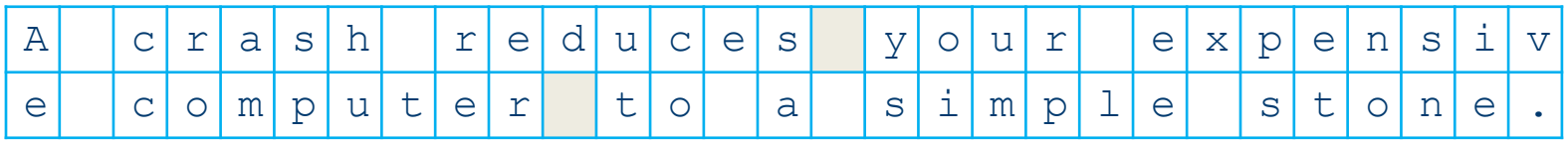
A crash reduces
your expensive computer
to a simple stone.

A		c	r	a	s	h		r	e	d	u	c	e	s		y	o	u	r		e	x	p	e	n	s	i	v
e		c	o	m	p	u	t	e	r		t	o		a		s	i	m	p	l	e		s	t	o	n	e	.

How to represent text?

1. Fixed-width records
2. Stream with embedded end-of-line markers

```
A crash reduces
your expensive computer
to a simple stone.
```

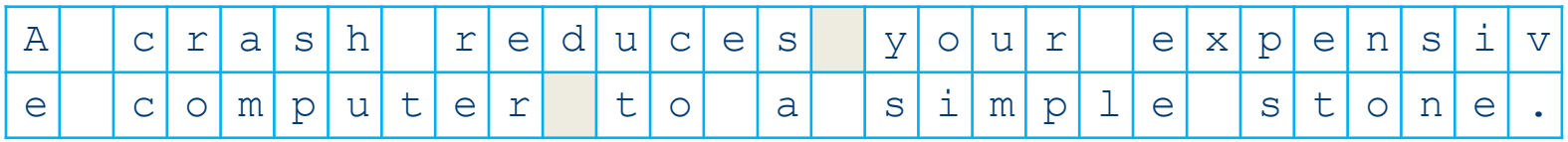


More flexible

How to represent text?

1. Fixed-width records
2. Stream with embedded end-of-line markers

```
A crash reduces
your expensive computer
to a simple stone.
```



More flexible

Wastes less space

How to represent text?

1. Fixed-width records
2. Stream with embedded end-of-line markers

```
A crash reduces
your expensive computer
to a simple stone.
```

A		c	r	a	s	h		r	e	d	u	c	e	s		y	o	u	r		e	x	p	e	n	s	i	v
e		c	o	m	p	u	t	e	r		t	o		a		s	i	m	p	l	e		s	t	o	n	e	.

More flexible

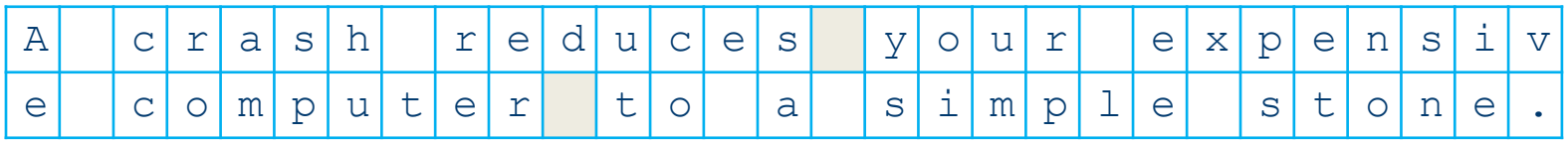
Skipping ahead is harder

Wastes less space

How to represent text?

1. Fixed-width records
2. Stream with embedded end-of-line markers

```
A crash reduces
your expensive computer
to a simple stone.
```



More flexible

Wastes less space

Skipping ahead is harder

What to use for end of line?

Unix: newline (`'\n'`)

Unix: newline (`'\n'`)

Windows: carriage return + newline (`'\r\n'`)

Unix: newline (`'\n'`)

Windows: carriage return + newline (`'\r\n'`)

Oh dear…

Unix: newline (`'\n'`)

Windows: carriage return + newline (`'\r\n'`)

Oh dear…

Python converts `'\r\n'` to `'\n'` and back on Windows

Unix: newline (`'\n'`)

Windows: carriage return + newline (`'\r\n'`)

Oh dear...

Python converts `'\r\n'` to `'\n'` and back on Windows

To prevent this (e.g., when reading image files)

open the file in *binary* mode

Unix: newline (`'\n'`)

Windows: carriage return + newline (`'\r\n'`)

Oh dear...

Python converts `'\r\n'` to `'\n'` and back on Windows

To prevent this (e.g., when reading image files)

open the file in *binary* mode

```
reader = open('mydata.dat', 'rb')
```

Back to characters…

Back to characters…

How to represent ĕ, β, Я, …?

Back to characters…

How to represent ĕ, β, Я, …?

7 bits = 0…127

Back to characters…

How to represent ĕ, β, Я, …?

7 bits = 0…127

8 bits (a byte) = 0…255

Back to characters…

How to represent ĕ, β, Я, …?

7 bits = 0…127

8 bits (a byte) = 0…255

Different companies/countries defined different meanings for 128…255

Back to characters…

How to represent ĕ, β, Я, …?

7 bits = 0…127

8 bits (a byte) = 0…255

Different companies/countries defined different meanings for 128…255

Did not play nicely together

Back to characters...

How to represent ĕ, β, Я, ...?

7 bits = 0...127

8 bits (a byte) = 0...255

Different companies/countries defined different meanings for 128...255

Did not play nicely together

And East Asian "characters" won't fit in 8 bits

1990s: Unicode standard

1990s: Unicode standard

Defines mapping from characters to integers

1990s: Unicode standard

Defines mapping from characters to integers

Does not specify how to store those integers

1990s: Unicode standard

Defines mapping from characters to integers

Does not specify how to store those integers

32 bits per character will do it…

1990s: Unicode standard

Defines mapping from characters to integers

Does not specify how to store those integers

32 bits per character will do it…

…but wastes a lot of space in common cases

1990s: Unicode standard

Defines mapping from characters to integers

Does not specify how to store those integers

32 bits per character will do it…

…but wastes a lot of space in common cases

Use in memory (for speed)

1990s: Unicode standard

Defines mapping from characters to integers

Does not specify how to store those integers

32 bits per character will do it…

…but wastes a lot of space in common cases

Use in memory (for speed)

Use something else on disk and over the wire

(Almost) everyone uses a variable-length encoding called UTF-8 instead

(Almost) everyone uses a variable-length encoding called UTF-8 instead

First 128 characters (old ASCII) stored in 1 byte each

(Almost) everyone uses a variable-length encoding called UTF-8 instead

First 128 characters (old ASCII) stored in 1 byte each

Next 1920 stored in 2 bytes, etc.

(Almost) everyone uses a variable-length encoding called UTF-8 instead

First 128 characters (old ASCII) stored in 1 byte each

Next 1920 stored in 2 bytes, etc.

`0xxxxxxx`	`7 bits`

(Almost) everyone uses a variable-length encoding called UTF-8 instead

First 128 characters (old ASCII) stored in 1 byte each

Next 1920 stored in 2 bytes, etc.

	`0xxxxxxx`	`7 bits`
`110yyyyy`	`10xxxxxx`	`11 bits`

(Almost) everyone uses a variable-length encoding called UTF-8 instead

First 128 characters (old ASCII) stored in 1 byte each

Next 1920 stored in 2 bytes, etc.

		0xxxxxxx	7 bits
	110yyyyy	10xxxxxx	11 bits
1110zzzz	10yyyyyy	10xxxxxx	16 bits

(Almost) everyone uses a variable-length encoding called UTF-8 instead

First 128 characters (old ASCII) stored in 1 byte each

Next 1920 stored in 2 bytes, etc.

			0xxxxxxx	7 bits
		110yyyyy	10xxxxxx	11 bits
	1110zzzz	10yyyyyy	10xxxxxx	16 bits
11110www	10zzzzzz	10yyyyyy	10xxxxxx	21 bits

(Almost) everyone uses a variable-length encoding called UTF-8 instead

First 128 characters (old ASCII) stored in 1 byte each

Next 1920 stored in 2 bytes, etc.

			0xxxxxxx	7 bits
		110yyyyy	10xxxxxx	11 bits
	1110zzzz	10yyyyyy	10xxxxxx	16 bits
11110www	10zzzzzz	10yyyyyy	10xxxxxx	21 bits

The good news is, you don't need to know

Python 2.* provides two kinds of string

Python 2.* provides two kinds of string

Classic: one byte per character

Python 2.* provides two kinds of string

Classic: one byte per character

Unicode: "big enough" per character

Python 2.* provides two kinds of string

Classic: one byte per character

Unicode: "big enough" per character

Write u'the string' for Unicode

Python 2.* provides two kinds of string

Classic: one byte per character

Unicode: "big enough" per character

Write u'the string' for Unicode

Must specify *encoding* when converting from Unicode to bytes

Python 2.* provides two kinds of string

Classic: one byte per character

Unicode: "big enough" per character

Write u'the string' for Unicode

Must specify *encoding* when converting from

Unicode to bytes

Use UTF-8

in Python 3.* strings are stored as Unicode by default

in Python 3.* strings are stored as Unicode by default

the default encoding for Python source code is UTF-8