GENERAL ASSEMBLY

MORE PYTHON

Dan Wilhelm, Lead Instructor
dan@danwilhelm.com

1) The built-in function `id` returns the memory address an object is stored at*.

```
a = [1, 'x']
a.append(a)
```

Using `id`, are `a[-1]` and `a` bound to the same object?

2) The built-in function `type` returns the data type (class) of an object. What are the types of the following?

```
a = (0, 1)      b = 0, 1        c = (0, 1,)
d = 0           e = (0)         f = (0,)
```

* In CPython only. The Python spec only requires that `id` return a unique identity for each object.

UPCOMING DATES

Friday, Oct. 19 - Unit 1 Quiz (in-class).

Data science workflow.

Basic command line, directories, paths.

Basic Python.

Friday, Oct. 26 - Unit 1 Project Due.

We will introduce the final project next month!

COURSE SCHEDULE: OFFICE HOURS

Office Hours (see #officehours on Slack)

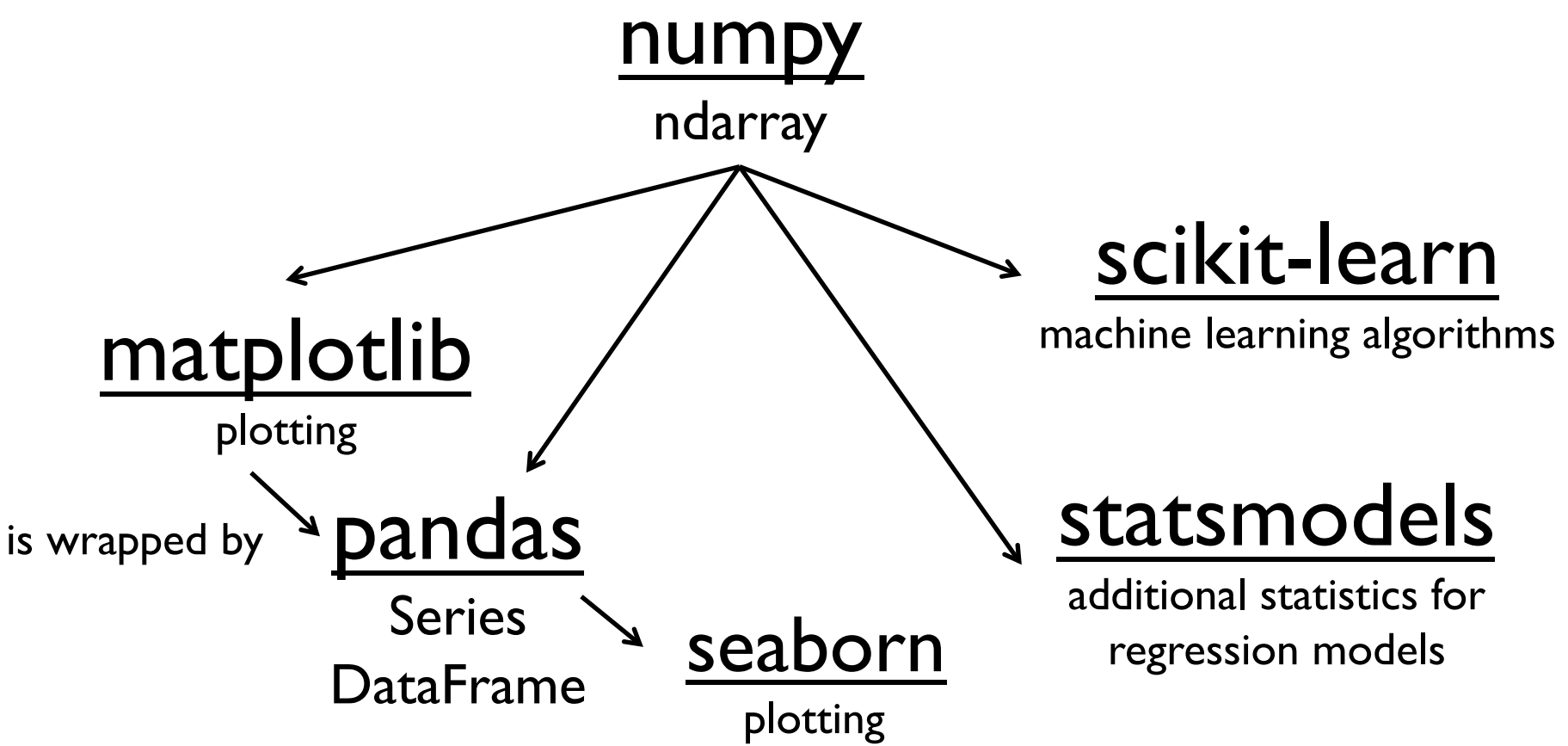
Tuesdays: 11am-12pm via https://davedoerner.youcanbook.me/

Wednesdays: 7pm-8pm via https://ginodefalco.youcanbook.me/

Fridays: 12pm-1pm (in class)

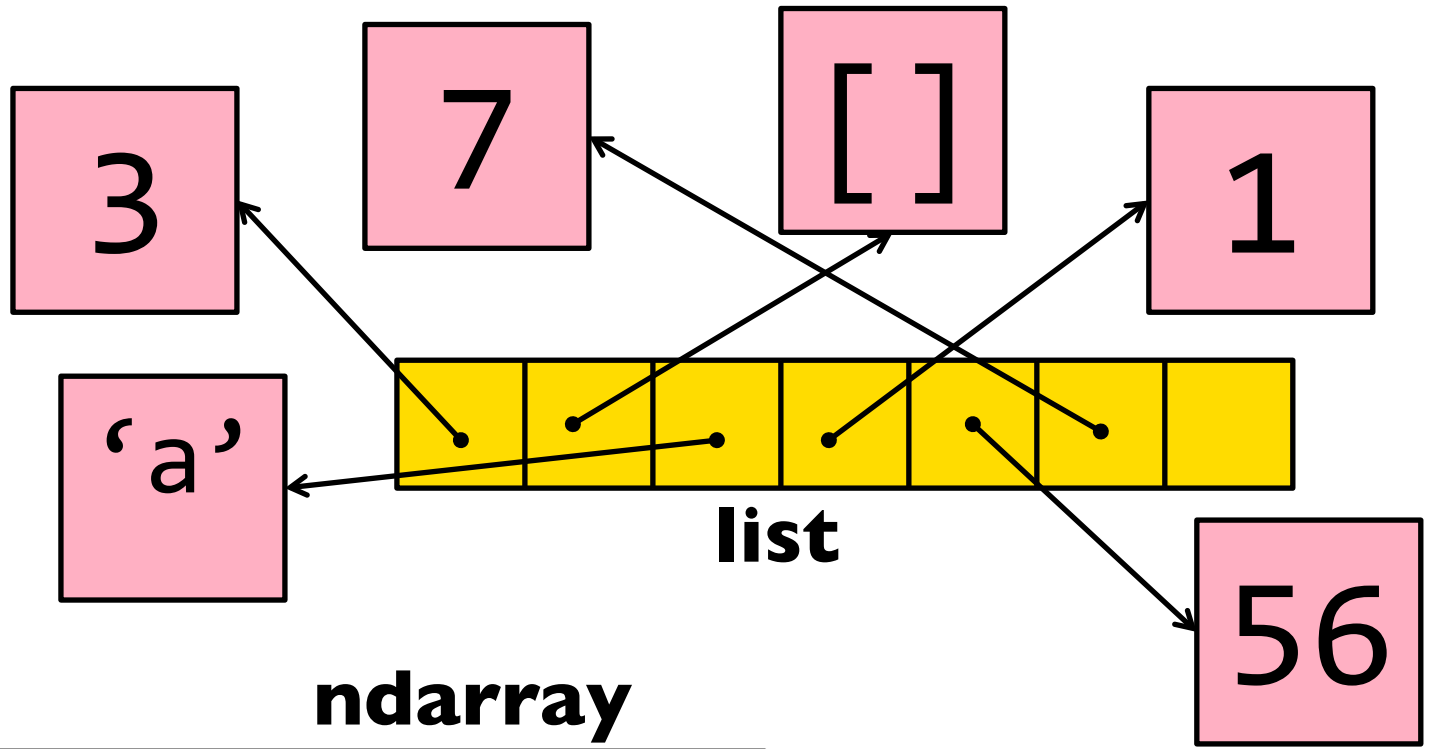
Questions during class? Feel free to Slack Gino, Dave, or #classroom.

Questions outside class? Feel free to Slack any of us, and we will get back with you when available.


numpy
ndarray
scikit-learn
machine learning algorithms
matplotlib
plotting
is wrapped by
pandas
Series
DataFrame
seaborn
plotting
statsmodels
additional statistics for
regression models

24 bytes/int
type (8)
ref count (8)
int (8)

scattered in memory (allows for heterogeneous types)



ndarray

3	12	84	1	56	7

contiguous in memory
8 bytes/int (int64)

- Items in Python lists are scattered in memory.

- Unlike a list, array items:
  - are the same data type
  - are stored contiguously in memory (great for caching)
  - cannot be arbitrarily removed or inserted
    (without shifting items over)
  - cannot exceed the allocated memory length
    (without allocating memory for a new array and copying all items)

- Array: **fixed-length**, ordered sequence of **homogeneous** items.
  - **Arrays are how computer memory actually works.**
  - *To insert:* Must allocate a new array then copy all items.
  - *To remove*: Must fill the removed item by copying all items over one.
  - *To extend*: Must allocate memory for a new array and copy all items.

- List: **unlimited-length**, ordered sequence of **heterogeneous** items.
  - Lists are abstractions – an easier way of thinking.
  - We assume insertion/deletion and extension can be done in constant time, but practically this may not always be true (depending on the implementation).

BITS AND BYTES

Bit

- Always in one of two states, typically referred to as 0 and 1.
- The atomic unit of memory.

Byte

- 8 bits
- The minimal addressable unit of memory.

BITS AND BYTES

- **1 bit:** can represent $2^1 = 2$ states (0 and 1).
- **2 bits:** can represent $2^2 = 4$ states (00, 01, 10, and 11).

We claim n bits can represent 2^n states.

- **n bits:** for each state in **(n-1) bits**, prepend a 0 or 1.
- Hence, there are twice as many states!
- And, $2*2^{n-1} = 2^n$.

```
00   -> 0(00) or 1(00)
01   -> 0(01) or 1(01)
10   -> 0(10) or 1(10)
11   -> 0(11) or 1(11)
```

YOUR MEMORY (AND DISK)

Examples of bits and bytes

```
np.array([1, 2, 3], np.int32)   # 32-bit integers

np.array([1, 2, 3], np.float64)   # 64-bit floats
```

YOUR MEMORY (AND DISK)

```
Memory:  01100001 01010011 10011111 01 …
Address: 0        1        2        3  …
```

YOUR MEMORY (AND DISK)

```
Hex:          0x61      0x53      0x9F
Decimal:       97        83       191
Memory:   01100001 01010011 10011111 01 …
Address:  0        1        2        3  …
```

YOUR MEMORY (AND DISK)

- All data is stored as 0s and 1s.
- So, we must come up with ways to encode different data types!

- **Unsigned integers** are encoded as binary (base 2).
- **Signed integers** are encoded using two's complement.
- **Floating point** is encoded using IEEE 754 (scientific notation).
- **Text** is encoded by mapping each possible state to a character.

YOUR MEMORY (AND DISK)

Examples of bits and bytes

```
np.array([1, 2, 3], np.int32)   # 32-bit integers

np.array([1, 2, 3], np.float64)   # 64-bit floats
```

LISTS VS TUPLES

- A list contains ordered data, typically of the same data type:

```
>>> x = [“Tim”, “Sandy”, “Martin”, “Shawna”]
>>> print(x[0])
```

- A tuple contains ordered groups of variables, often of different data types:

```
>>> x = (“Tim”, 5)   # (name, age) describe one person
>>> print(x)
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
```

LISTS VS TUPLES

- Lists: Mutable (can be altered), typically homogenous values
- Tuples: Immutable, typically groups of items that go together

```
>>> people = [('Tim', 32), ('Sandy', 45)]
>>> points = [(0,0,0), (5,4,1), (7,7,8)]
>>> menu_item, price = ('Burger', 2.99)   # unpacking
>>> def max_population(cities):   # multiple return values
        ...
        return (city_name, population)
```

STRING FORMATTING

```
“Person #{} is {}”.format(2, ‘George’)

“Person #{num} is {name}”.format(num=5, name=‘Henry’)

“The price is ${price:.2f}.”.format(price=4.5127)

price = 4.5127
f“The price of two items is {(price * 2):.2f} dollars.”
```

MODULES

```
import math
>>> math.sqrt(5)

from math import sqrt
>>> sqrt(5)

from math import *
>>> sqrt(5)
>>> tan(5)
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100)]
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    if num % 2 == 0:
        cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100) if num % 2 == 0]
```

NOTE you can use these for filtering!

FILES & UNICODE

FILES (OLD WAY)

```
file_in = open('test.txt', 'r', encoding='utf-8')

for line in file_in:
    print(line)

file_in.close()
```

FILES (OLD WAY)

```
file_in = open('test.txt', 'r', encoding='utf-8')


for line in file_in:
    print(line)


file_in.close()
```

common file modes

'r' – read
'w' – write
'a' – append
'b' – binary

FILES (NEW WAY) – READING

```
# ‘with’ insures the file is closed if an exception occurs
with open(‘test.txt’, ‘r’ , encoding=‘utf-8’) as fin:
    for line in fin:
        print(line)

# Same thing, except the ‘with’ block ensures the file is
# closed – even if an exception occurs!
```

FILES (NEW WAY) – WRITING

```
# Write list of strings ‘lines’ to the file
with open(‘test.txt’, ‘w’) as fout:
    for line in lines:
       fout.write(line + “\n”)
```

ENCODINGS

- All files are just sequences of bytes (aka 8-bit numbers). So, all files look alike.
- Hence, the operating system needs a way to know how to interpret a file's contents.

ENCODINGS

- All files are just sequences of bytes (aka 8-bit numbers). So, all files look alike.
- Hence, the operating system needs a way to know how to interpret a file's contents.
- There are only two main ways to do this:
  1. The file extension provides a hint at how to interpret the file, e.g.:

     .jpeg .htaccess .txt .docx
  2. The beginning of the file could provide clues, e.g.:

     all JPEG files start with the sequence: 255, 216

ASCII VS UNICODE

- Text files (.txt, .py, .html) have historically used the 7-bit **ASCII** encoding
- However, ASCII only supports the English alphabet. So, many other text encodings were created.
- Collectively, these alternative encodings are known as **Unicode**.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

ASCII

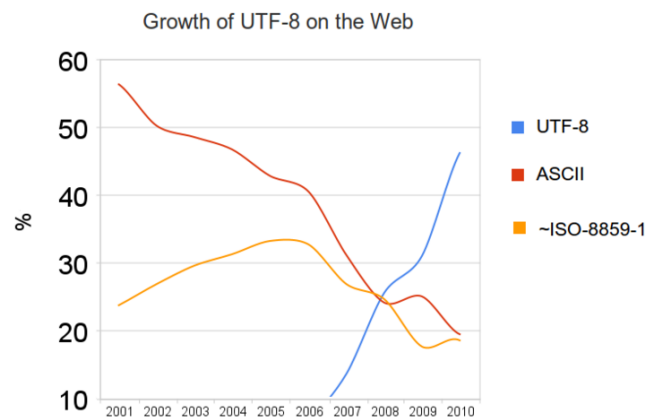
A 7-bit text encoding

ASCII VS UNICODE

- *Problem*: Nowadays, text files could be encoded via ASCII or one of many Unicode encodings.
- However, there is no way to infer how to interpret the numbers -- the filename extensions are all the same, and text files immediately begin with the text content!

ASCII VS UNICODE

- For example, the most popular encoding is UTF-8. It relies on ASCII being 7 bits and uses the 8th bit as a flag to indicate a special character.
- Hence, it is backwards compatible with ASCII. An ASCII file is also a UTF-8 file! (Assuming the 8th bits are all zero.)



ASCII VS UNICODE

- For example, the most popular encoding is UTF-8. It relies on ASCII being 7 bits and uses the 8th bit as a flag to indicate a special character.
- Hence, it is backwards compatible with ASCII. An ASCII file is also a UTF-8 file! (Assuming the 8th bits are all zero.)
- If an ASCII file has a non-zero 8th bit, this will throw a Python exception saying it is invalid ASCII!
- If a UTF-8 encoded-file does not follow the UTF-8 rules, then it will also throw a Python exception, e.g. if Byte 1 is 110xxxxx, the next byte *must be* 10xxxxxx, where ‘x’ is any bit.

COMMON UNICODE ENCODINGS FOR OPEN()

‘ascii’	ASCII
‘latin-1’ (‘iso-8859-1’)	ASCII w/ extended ASCII
‘utf-8’	variable-length encoding (min. 1 byte)*
‘utf-16’	var-length encoding (min. 2 bytes)

* backward-compatible with ASCII

Also:

- \u<CODE POINT> is the escape char for 2-byte Unicode chars
- \U<CODE POINT> for 4-byte Unicode chars

UTF-8: A VARIABLE-LENGTH ENCODING

How to store different Unicode code words

	1st byte	2nd byte	3rd byte
U+0000 – U+007F	**0nnn nnnn**		
U+007F – U+07FF	110n nnnn	10nn nnnn	
U+0800 – U+FFFF	1110 nnnn	10nn nnnn	10nn nnnn

Note the backward-compatibility with ASCII! (bolded)

ASCII VS UNICODE

- If we ask Python to interpret a file using a certain encoding and the file does not follow these rules, then an exception will be thrown.
- This is unfortunately the only way to know whether we guess the encoding correctly – attempt to interpret the file numbers in one way and see if it follows the encoding's rules.

ASCII VS UNICODE

- If we ask Python to interpret a file using a certain encoding and the file does not follow these rules, then an exception will be thrown.
- This is unfortunately the only way to know whether we guess the encoding correctly – attempt to interpret the file numbers in one way and see if it follows the encoding's rules.
- So, it is preferred to try opening the file using different encoding. If this does not work, you can choose to ignore errors.
- Note: When you open a file in an IDE or text editor and save it, the IDE may resave the file in a different encoding!

MODULES

```
import math
>>> math.sqrt(5)

from math import sqrt
>>> sqrt(5)

from math import *   # NOT RECOMMENDED – POLLUTES NAMESPACE
>>> sqrt(5)
>>> tan(5)
```

LITERALS

Below, we are creating objects without assigning names. These are called **literals**:

- 3 # int object
- [1, 2, 3] # list object
- {1, 2, 3} # set object
- (1, 2, 3) # tuple object

LITERALS

Below, we are creating objects without assigning names. These are called **literals**:

- 3 # int object
- [1, 2, 3] # list object
- {1, 2, 3} # set object
- (1, 2, 3) # tuple object

Can we do the same with functions?

Note that **def f(x)** auto-creates a name **f** in the namespace.

LAMBDA FUNCTIONS

def inc(x): return x + 1

- Creates a function object.
- Assigns the name ‘inc’ to the object.

lambda x: x + 1

- Only creates a function object.
- Intended for a single line.
- Implicitly returns the last evaluated expression.

LAMBDA FUNCTIONS

def inc(x): return x + 1

- Creates a function object.
- Assigns the name ‘inc’ to the object.

lambda x: x + 1

- Only creates a function object.
- Intended for a single line.
- Implicitly returns the last evaluated expression.

pow(x, y): return x**y

lambda x, y: x**y

LAMBDA FUNCTIONS

inc(2)

(inc)(2) **# (x) means evaluate x first, e.g. 5 * (3 + 4)**

- Evaluating the name inc results in a function object.

(lambda x: x + 1)(2)

- Evaluating the lambda results in a function object.

Interestingly, inc(2) is syntactic sugar for inc.__call__(2)

LAMBDA FUNCTIONS: USES

```
# Square each element of a list.
numbers = [0, 1, 2, 3, 4, 5]
list(map(lambda n: n*n, numbers))

# Sort by ages
people = [['Mike', 40], ['Terry', 20], ['Sarah', 30]]
sorted(people, key=lambda x: x[1])
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100)]
```

LIST COMPREHENSIONS

```
cubes = []
for num in range(100):
    if num % 2 == 0:
        cubes.append(num**3)
```

BECOMES

```
cubes = [num**3 for num in range(100) if num % 2 == 0]
```

NOTE you can use these for filtering!

COLUMNAR DATA

```
people = [('Tim', 20),
          ('Sally', 22),
          ('Ryan', 25)]
```

How do we get the first **column** (i.e. a list of names)?

COLUMNAR DATA

```
people = [('Tim', 20),
          ('Sally', 22),
          ('Ryan', 25)]
```

How do we get the first **column** (i.e. a list of names)?

```
names = [person[0] for person in people]
```

ZIP

- Let's go in the opposite direction. We have separate lists of names and ages.
- How do we combine them element-by-element to make a single list of tuples? The easy way is to use zip, which combines each of the first elements, each of the second elements, etc.:

```
names = [‘Tim’, ‘Sally’, ‘Ryan’]

ages = [20, 22, 25]

list(zip(names, ages))

>> [(‘Tim’, 20), (‘Sally’, 22), (‘Ryan’, 25)]
```

FUNCTIONS

- A function allows us to take complex code and refer to it in an easy way, reducing the complexity in our minds.
- For example, "x % 2 == 1" can be tough to understand for beginners. Hence, to make the program easier to read, we refer to what the code does in English, as part of a function call that returns a value:

```
>>> def is_odd(num):
...     return (num % 2 == 1)
```

```
num = 13195

factors = []
for n in range(1,num+1):
        if num % n == 0:
                factors.append(n)

prime_factors = []
for factor in factors:
        n = 1
        while n < factor:
                n += 1
                if factor % n == 0:
                        break
        if n == factor:
                prime_factors.append(factor)

print(max(prime_factors)) # should be 29
```

Hard to Read!

```
def is_multiple(n, m):
        """ Is n a multiple of m? """
        return n % m == 0

def get_factors(num):
        return [n for n in range(1,num+1) if is_multiple(num, n)]

def is_prime(num):
        return True if len(get_factors(num)) == 2 else False

def prime_factors(num):
        return [f for f in get_factors(num) if is_prime(f)]

print(max(prime_factors(13195)))     # should be 29
```

```
# TESTS
print("prime? 2, 3, 4, 41, 111 => ",
      is_prime(2), is_prime(3), is_prime(4),
      is_prime(41), is_prime(111))
print("factors of 36 (contains 6?) => ", get_factors(36))
print("factors of 13195 => ", get_factors(13195))
print("prime factors of 13195 => ", prime_factors(13195))
```

CODING CHALLENGE!

INTERACTING WITH THE COMMAND LINE

- Retrieving command-line arguments is easy!

```
import sys
sys.argv    # ARGument Vector (list)

Example: python test.py 1 2 3
sys.argv: ['test.py', '1', '2', '3']
```

INTERACTING WITH THE COMMAND LINE

- Reading from stdin is just like reading a file!

```
import sys
for line in sys.stdin:
    print(line.strip())
```

FILES

```
# ‘with’ ensures the file is closed if an exception occurs
with open(‘test.txt’, ‘r’) as fin:
    for line in fin:
        print(line)

# Write list of strings ‘lines’ to the file
with open(‘test.txt’, ‘w’) as fout:
    for line in lines:
       fout.write(line + “\n”)
```

QUOTES

- ‘vs “ ← same – both support escape characters (“\n“)
- “”” ← triple quotes – allows actual newlines

- \ ← if at the end of a non-quoted line of code, allows you to split the line of code (there is an invisible newline after it)

```
>>> names = “Mike Wallace\nClara Simmons”
>>> names.split(“\n”)
>>> names.replace(“\n”, “, “)
```

ENUMERATE – WHEN YOU NEED A LOOP INDEX

```
index = 0
for person in people:
        print(“Person #{} is {}”.format(index, person))
        index += 1
```

BECOMES

```
for index,person in enumerate(people):
        print(“Person #{} is {}”.format(index, person))
```

DATES AND TIMES

- `datetime.date -> year, month, day`
- `datetime.time -> hour, minute, second, microsecond, tzinfo (TimeZone INFO)`
- `datetime.datetime -> year, month, day, hour, minute, second, microsecond, tzinfo`

- `datetime.timedelta – difference between dates/times`

EXCEPTIONS

```
try:
        num = int('not an int')
except:   # catches ALL exceptions
        print('Exception caught!')


try:
        num = int('not an int')
except ValueError:   # catches the ValueError exception
        print('Exception caught!')
```

EXERCISES!