UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Lambdas
Environments
Dictionaries

Computing In the News: Dark Patterns

FTC Report Shows Rise in Sophisticated Dark Patterns Designed to Trick and Trap Consumers 9/15/2022 (See also darkpatterns.org)

Tactics Include Disguised Ads, Difficult-to-Cancel Subscriptions, Buried Terms, and Tricks to Obtain Data

The Federal Trade Commission released a report today showing how companies are increasingly using sophisticated design practices known as “dark patterns” that can trick or manipulate consumers into buying products or services or giving up their privacy. The dark pattern tactics detailed in the report include disguising ads to look like independent content, making it difficult for consumers to cancel subscriptions or charges, burying key terms or junk fees, and tricking consumers into sharing their data. The report highlighted the FTC’s efforts to combat the use of dark patterns in the marketplace and reiterated the agency’s commitment to taking action against tactics designed to trick and trap consumers.

Announcements

- **Late Adds / Transfers**
  - Form out in a day or two
- Schedule Updates + Changes
  - Likely add a Friday afternoon lab
  - Will likely add Tues or Weds afternoon
  - A couple TAs times will shift
  - Office Hours may move

UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Lambda Expressions

Learning Objectives

- Lambda are anonymous functions, which use expressions
  - Don't use `return`, lambdas always return the value of the expression.
  - They are typically short and concise
  - They don't have an "intrinsic" name when using an environment diagram.
    - Their name is the character λ

lambda

- Function expression
  - "anonymous" function creation
  - Expression, not a statement, no return or any other statement

lambda <arg or arg_tuple> : <expression using args>

```
add_one = lambda v : v + 1
```

```
def add_one(v):
    return v + 1
```

Lambdas

```
>>> def inc_maker(i):
...     return lambda x: x+i
...
>>> inc_maker(3)
<function inc_maker.<locals>.<lambda> at 0x10073c510>

>>> inc_maker(3)(4)
7
>>> map(inc_maker(3), [1,2,3,4])
<map object at 0x1020950b8>

>>> list(map(inc_maker(3), [1,2,3,4]))
[4, 5, 6, 7]
>>>
```

Lambda with HOFs

- **A function that returns (makes) a function**

```
def leq_maker(c):
    return lambda val: val <= c
```

```
>>> leq_maker(3)
<function leq_maker.<locals>.<lambda> at 0x1019d8c80>

>>> leq_maker(3)(4)
False

>>> filter(leq_maker(3), [0,1,2,3,4,5,6,7])
[0, 1, 2, 3]
```

Lambda Examples

```
>>> sorted([1,2,3,4,5], lambda x: x)
    [1, 2, 3, 4, 5]

>>> sorted([1,2,3,4,5], lambda x: -x)
    [5, 4, 3, 2, 1]

>>> sorted([(2, "hi"), (1, "how"), (5, "goes"), (7, "I")],
           lambda x:x[0])
[(1, 'how'), (2, 'hi'), (5, 'goes'), (7, 'I')]

>>> sorted([(2, "hi"), (1, "how"), (5, "goes"), (7, "I")],
          lambda x:x[1])
    [(7, 'I'), (5, 'goes'), (2, 'hi'), (1, 'how')]

>>> sorted([(2,"hi"),(1,"how"),(5,"goes"),(7,"I")],
          lambda x: len(x[1]))
    [(7, 'I'), (2, 'hi'), (1, 'how'), (5, 'goes')]
```

UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Environment Diagrams

Python Tutor Examples: compose

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add_2 = make_adder(2)
add_3 = make_adder(3)
x = add_2(5)
y = add_3(x)
```

Environment Diagrams

- Organizational tools that help you understand code
- **Terminology:**
  - **Frame:** keeps track of variable-to-value bindings, each function call has a frame
  - **Global Frame:** global for short, the starting frame of all python programs, doesn't correspond to a specific function
  - **Parent Frame:** The frame of where a function is defined (default parent frame is global)
  - **Frame number:** What we use to keep track of frames, f1, f2, f3, etc
  - **Variable** vs **Value**: x = 1. x is the **variable**, 1 is the **value**

Environment Diagrams Reminders

1. Always draw the global frame first
2. When evaluating assignments (lines with single equal), always evaluate right side first
3. When you CALL a function MAKE A NEW FRAME!
4. When assigning a primitive expression (number, boolean, string) write the value in the box
5. When assigning anything else (lists, functions, etc.), draw an arrow to the value
6. When calling a function, name the frame with the intrinsic name – the name of the function that variable points to
7. The parent frame of a function is the frame in which it was defined in (default parent frame is global)
8. If the value for a variable doesn’t exist in the current frame, search in the parent frame

Demo

Example 1:

- Primitives and Functions: Environment Diagram Python Tutor:

Example 2:

- make_adder Higher Order Function: Environment Diagram Python Tutor Link

Example 3:

- Compose Python Tutor Link

Example 1

```
a = "chipotle"
b = 5 > 3
c = 8

def foo(c):
    return c - 5

def bar():
    if b:
        a = "taco bell"

result1 = foo(10)
result2 = bar()
```

Example 2

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

n = 10
add_2 = make_adder(2)
x = add_2(5)
```

Python Tutor Examples

```
add_2 = make_adder(2)
add_3 = make_adder(3)

x = add_2(2)
def compose(f, g):
    def h(x):
        return f(g(x))
    return h

add_5 = compose(add_2, add_3)
z = add_5(x)
```

Environment Diagram Tips / Links

- NEVER draw an arrow from one variable to another.
- Useful Resources:
  - http://markmiyashita.com/cs61a/environment_diagrams/rules_of_environment_diagrams/
  - http://albertwu.org/cs61a/notes/environments.html

Computational Structures in Data Science

UC Berkeley EECS
Lecturer
Michael Ball

Dictionaries

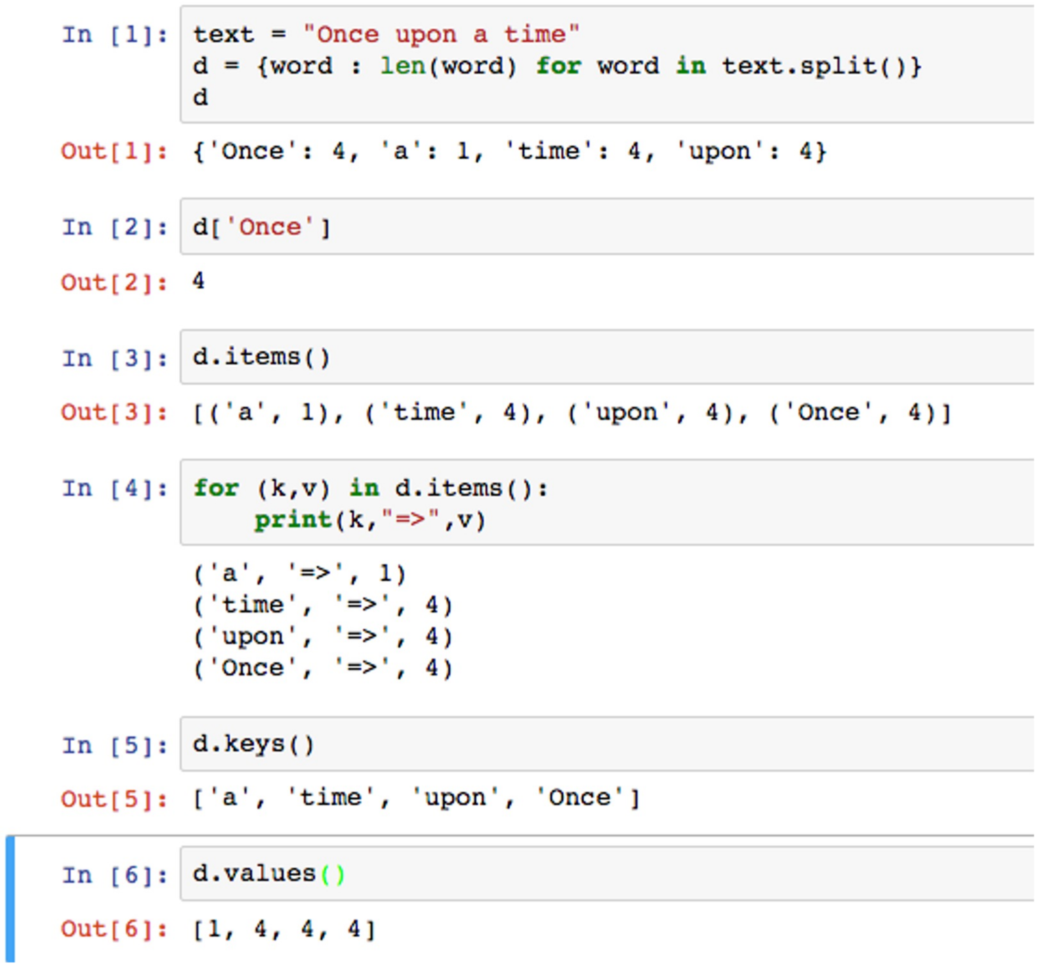
Learning Objectives

- Dictionaries are a new type in Python
- Lists let us index a value by a number, or position.
- Dictionaries let us index data by other kinds of data.

Dictionaries

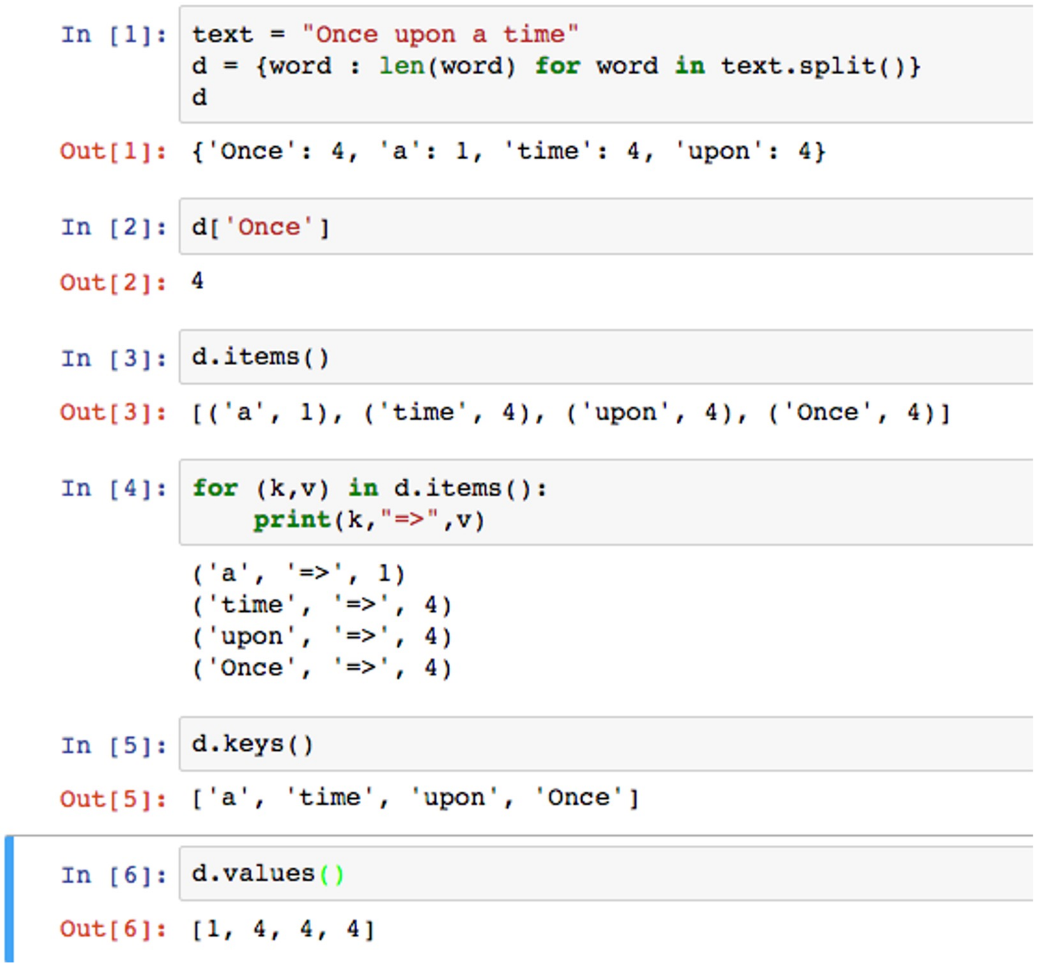
- Constructors:
  - » `dict( <list of 2-tuples> )`
  - » `dict( <key>=<val>, ...) # like kwargs`
  - » `{ <key exp>:<val exp>, …  }`
  - » `{ <key>:<val> for <iteration expression> }`
    - `>>> {x:y for x,y in zip(["a","b"],[1,2])}`
    - `{'a': 1, 'b': 2}`
- Selectors: `<dict> [ <key> ]`
  - » <dict>.keys(), .items(), .values()
  - » <dict>.get(key [, default])
- Operations:
  - » Key in, not in, len, min, max
  - » <dict>[<key>] = <val>

Dictionary Example



```
In [1]:  text = "Once upon a time"
         d = {word : len(word) for word in text.split()}
         d

Out[1]:  {'Once': 4, 'a': 1, 'time': 4, 'upon': 4}

In [2]:  d['Once']

Out[2]:  4

In [3]:  d.items()

Out[3]:  [('a', 1), ('time', 4), ('upon', 4), ('Once', 4)]

In [4]:  for (k,v) in d.items():
             print(k,"=>",v)

         ('a', '=>', 1)
         ('time', '=>', 4)
         ('upon', '=>', 4)
         ('Once', '=>', 4)

In [5]:  d.keys()

Out[5]:  ['a', 'time', 'upon', 'Once']

In [6]:  d.values()

Out[6]:  [1, 4, 4, 4]
```

Dictionary Example



```
In [1]:  text = "Once upon a time"
         d = {word : len(word) for word in text.split()}
         d

Out[1]:  {'Once': 4, 'a': 1, 'time': 4, 'upon': 4}

In [2]:  d['Once']

Out[2]:  4

In [3]:  d.items()

Out[3]:  [('a', 1), ('time', 4), ('upon', 4), ('Once', 4)]

In [4]:  for (k,v) in d.items():
             print(k,"=>",v)

         ('a', '=>', 1)
         ('time', '=>', 4)
         ('upon', '=>', 4)
         ('Once', '=>', 4)

In [5]:  d.keys()

Out[5]:  ['a', 'time', 'upon', 'Once']

In [6]:  d.values()

Out[6]:  [1, 4, 4, 4]
```