



# DM536

## Introduction to Programming

Peter Schneider-Kamp

[petersk@imada.sdu.dk](mailto:petersk@imada.sdu.dk)

<http://imada.sdu.dk/~petersk/DM536/>

# Additional Project Support Hours

- get help with your project on Wednesday 08-12
- TAs will be in or around IMADA terminal room
- code word “Slartibartfast”
- Until then ...
- ... go and DO something! 😊

# DICTIONARIES

# Generalized Mappings

- list = mapping from integer indices to values
- dictionary = mapping from (almost) any type to values
- indices are called *keys* and pairs of keys and values *items*
- empty dictionaries created using curly braces “{}”
- Example: `en2da = {}`
- keys are assigned to values using same syntax as for sequences
- Example: `en2da["queen"] = "dronning"`  
`print en2da`
- curly braces “{” and “}” can be used to create dictionary
- Example: `en2da = {"queen" : "dronning", "king" : "konge"}`

# Dictionary Operations

- printing order can be different: `print en2da`
- access using indices: `en2da["king"] == "konge"`
- `KeyError` when key not mapped: `print en2da["prince"]`
- length is number of items: `len(en2da) == 2`
- `in` operator tests if key mapped: `"king" in en2da == True`  
`"prince" in en2da == False`
- `keys()` method gives list of keys:  
`en2da.keys() == ["king", "queen"]`
- `values()` method gives list of values:  
`en2da.values() == ["konge", "dronning"]`
- useful e.g. for test if value is used:  
`"prins" in en2da.values() == False`

# Dictionaries as Sets

- dictionaries can be used as sets
- **Idea:** assign `None` to all elements of the set
- Example: representing the set of primes smaller than 20  
`primes = {2: None, 3: None, 5: None, 7: None, 11: None, 13: None, 17: None, 19: None}`
- then `in` operator can be used to see if value is in set
- Example:  
`15 in primes == False`  
`17 in primes == True`
- for lists, needs steps proportional to number of elements
- for dictionary, needs (almost) constant number of steps

# Counting Letter Frequency

- **Goal:** count frequency of letters in a string (*histogram*)
- many possible implementations, e.g.:
  - create 26(+3?) counter variables for each letter!; use chained conditionals (`if ... elif ... elif ...`) to increment
  - create a list of length 26(+3?); increment the element at index  $n-1$  if the  $n$ -th letter is encountered
  - create a dictionary with letters as keys and integers as values; increment using index access
- all these implementations work (differently)
- big differences in *runtime* and *ease of implementation*
- choice of data structure is a *design decision*

# Counting with Dictionaries

- fast and counts all characters – no need to fix before!

```
def histogram(word):
```

```
    d = {}
```

```
    for char in word:
```

```
        if char not in d:
```

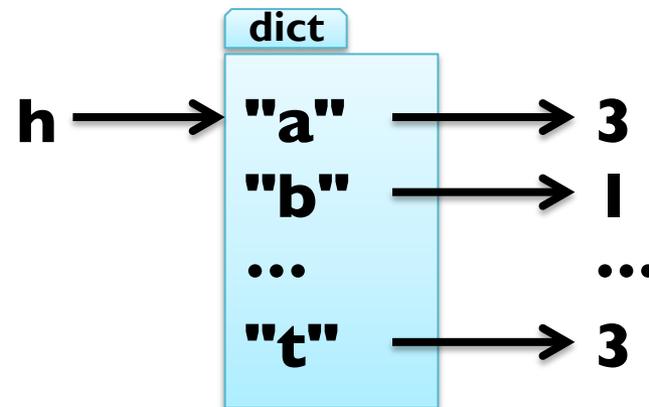
```
            d[char] = 1
```

```
        else:
```

```
            d[char] += 1
```

```
    return d
```

- Example: `h = histogram("slartibartfast")`  
`h == {"a":3, "b":1, "f":1, "i":1, "l":1, "s":2, "r":2, "t":3}`



# Counting with Dictionaries

- fast and counts all characters – no need to fix before!

```
def histogram(word):
```

```
    d = {}
```

```
    for char in word:
```

```
        if char not in d:
```

```
            d[char] = 1
```

```
        else:
```

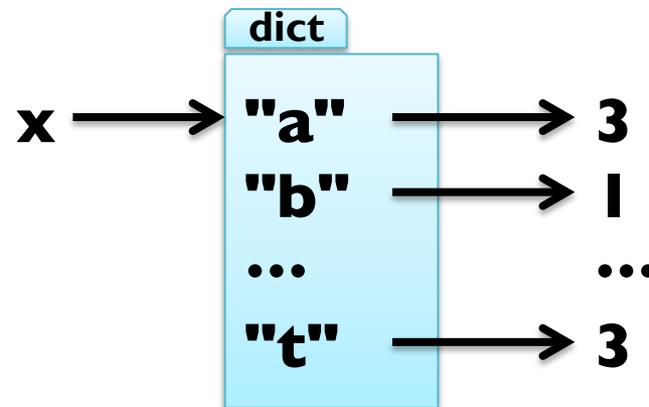
```
            d[char] += 1
```

```
    return d
```

- access using the `get(k, d)` method:

```
h.get("t", 0) == 3
```

```
h.get("z", 0) == 0
```



# Traversing Dictionaries

- using a `for` loop, you can traverse all keys of a dictionary

- Example: 

```
for key in en2da:  
    print key, en2da[key]
```

- you can also traverse all values of a dictionary

- Example: 

```
for value in en2da.values():  
    print value
```

- finally, you can traverse all items of a dictionary

- Example: 

```
for item in en2da.items():  
    print item[0], item[1]      # key, value
```

# Reverse Lookup

- given dict. `d` and key `k`, finding value `v` with `v == d[k]` easy
- this is called a dictionary *lookup*
- given dict. `d` and value `v`, finding key `k` with `v == d[k]` hard
- there might be more than one key mapping to `v` (cf. example)
- Possible implementation 1:

```
def reverse_lookup(d, v):  
    result = []  
    for key in d:  
        if d[key] == v:  
            result.append(key)  
    return result
```

- returns empty list, when no key maps to value `v`

# Reverse Lookup

- given dict.  $d$  and key  $k$ , finding value  $v$  with  $v == d[k]$  easy
- this is called a dictionary *lookup*
- given dict.  $d$  and value  $v$ , finding key  $k$  with  $v == d[k]$  hard
- there might be more than one key mapping to  $v$  (cf. example)
- Possible implementation 2:

```
def reverse_lookup(d, v):
```

```
    for k in d:
```

```
        if d[k] == v:
```

```
            return k
```

```
    raise ValueError
```

- gives error when no key maps to value  $v$

# Reverse Lookup

- given dict.  $d$  and key  $k$ , finding value  $v$  with  $v == d[k]$  easy
- this is called a dictionary *lookup*
- given dict.  $d$  and value  $v$ , finding key  $k$  with  $v == d[k]$  hard
- there might be more than one key mapping to  $v$  (cf. example)
- Possible implementation 2:

```
def reverse_lookup(d, v):  
    for k in d:  
        if d[k] == v:  
            return k  
    raise ValueError, "value not found in dictionary"
```

- gives error when no key maps to value  $v$

# Dictionaries and Lists

- lists cannot be keys, as they are mutable
- list can be values stored in dictionaries
- Example: inverting a dictionary

```
def invert_dict(d):
```

```
    inv = {}
```

```
    for key in d:
```

```
        val = d[key]
```

```
        if val not in inv:
```

```
            inv[val] = [key]
```

```
        else:
```

```
            inv[val].append(key)
```

```
    return inv
```

# Dictionaries and Lists

- lists cannot be keys, as they are mutable
- list can be values
- Example: inverting a dictionary

```
def invert_dict(d):
```

```
    inv = {}
```

```
    for key in d:
```

```
        val = d[key]
```

```
        if val not in inv:
```

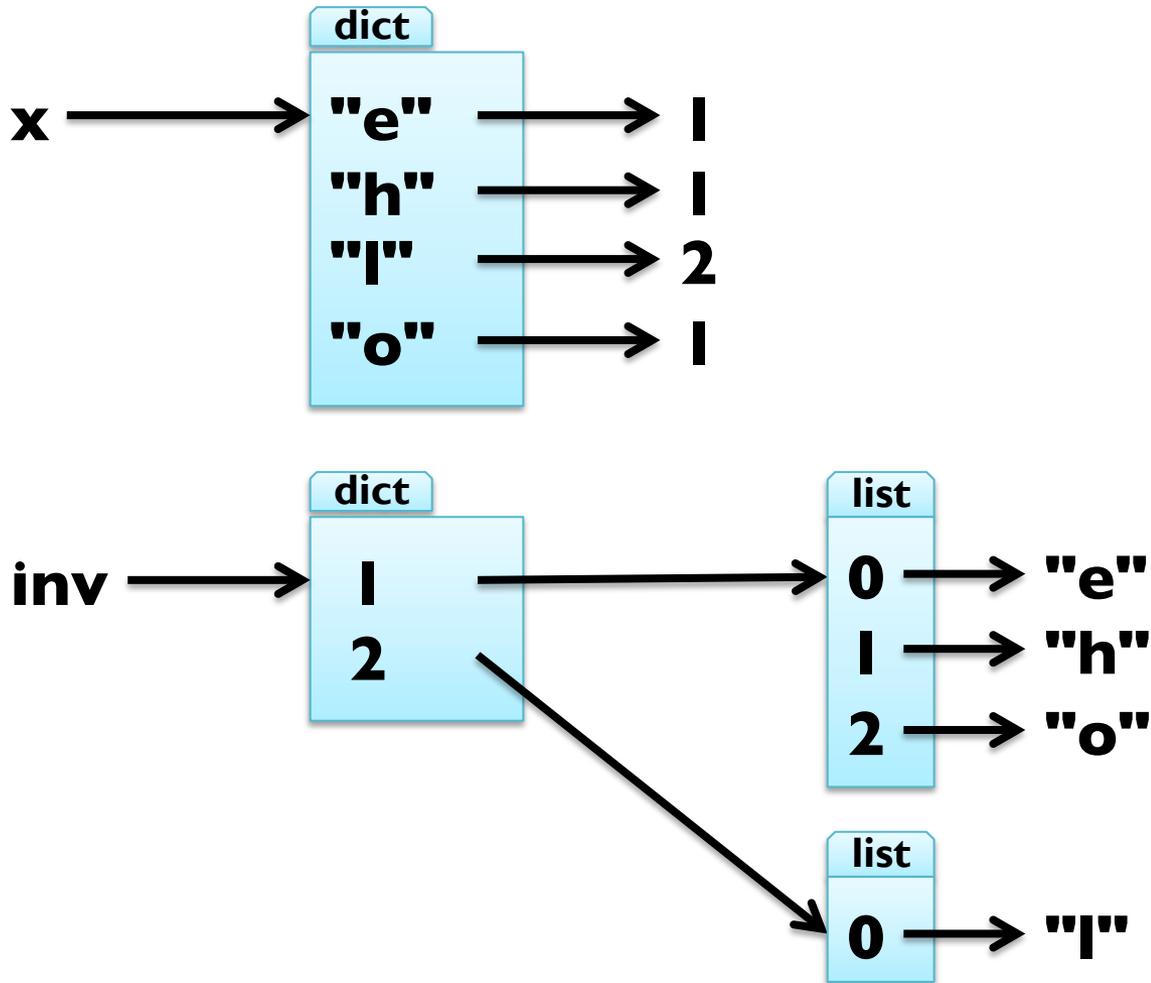
```
            inv[val] = []
```

```
            inv[val].append(key)
```

```
    return inv
```

- Example: `print invert_dict(histogram("hello"))`

# Dictionaries and Lists



- Example: `print invert_dict(histogram("hello"))`

# Memoizing

- Fibonacci numbers lead to exponentially many calls:

```
def fib(n):
```

```
    if n in [0,1]: return n
```

```
    return fib(n-1) + fib(n-2)
```

- keeping previously computed values (*memos*) helps:

```
known = {0:0, 1:1}
```

```
def fib_fast(n):
```

```
    if n in known:
```

```
        return known[n]
```

```
    res = fib_fast(n-1) + fib_fast(n-2)
```

```
    known[n] = res
```

```
    return res
```

# Global Variables

- known is created outside `fib_fast` and belongs to `__main__`
- such variables are called *global*
- many uses for global variables (besides memoization)
- Example 1: flag for controlling output

```
debug = True
```

```
def pythagoras(a,b):
```

```
    if debug:    print "pythagoras with a =d", a, " and b = d", b
```

```
    result = math.sqrt(a**2 + b**2)
```

```
    if debug:    print "result of pythagoras:", result
```

```
    return result
```

# Global Variables

- known is created outside `fib_fast` and belongs to `__main__`
- such variables are called *global*
- many uses for global variables (besides memoization)
- Example 2: track number of calls

```
num_calls = 0
```

```
def pythagoras(a,b):
```

```
    global num_calls
```

```
    num_calls += 1
```

```
    return math.sqrt(a**2 + b**2)
```

- gives `UnboundLocalError` as `num_calls` is local to `pythagoras`
- declare `num_calls` to be global using a `global` statement

# Long Integers

- Python uses 32 or 64 bit for `int`
- this limits the numbers that can be represented:
  - 32 bit: from  $-2^{31}$  to  $2^{31}-1$
  - 64 bit: from  $-2^{63}$  to  $2^{63}-1$
- for larger numbers, Python automatically uses `long` integers
- Example:

```
fib(93) == 12200160415121876738L
```

- `long` integers work just like `int`, only with "L" as suffix

- Example:  $2^{64} + 2^{64} == 2^{65}$

```
fib(100)**fib(20) # has 139016 digits :-o
```

# Debugging Larger Datasets

- debugging larger data sets, simple printing can be too much
  1. scale down the input – start with the first n lines; a good value for n is a small value that still exhibits the problem
  2. scale down the output – just print a part of the output; when using strings and lists, slices are very handy
  3. check summaries and types – check that type and len(...) of objects is correct by printing them instead of the object
  4. write self-checks – include some *sanity checks*, i.e., test Boolean conditions that should definitely hold
  5. pretty print output – even larger sets can be easier to interpret when printed in a more human-readable form

# TUPLES

# Tuples as Immutable Sequences

- tuple = immutable sequence of values
- like lists, tuples are indexed by integers
- tuples can be enclosed in parentheses “(” and “)”
- Example:

```
t1 = "D", "o", "u", "g", "l", "a", "s"  
t2 = (65, 100, 97, 109, 115)  
t3 = 42,      # or (42,) - but not (42)
```
- tuples can be created from sequences using `tuple(iterable)`
- Example:

```
t1 == tuple("Douglas")  
tuple(["You", 2]) == ("You", 2)
```

# Tuples as Immutable Sequences

- tuple = immutable sequence of values
- like lists, tuples are indexed by integers
- tuples can be accessed using indices and slices
- Example:

```
t = "D", "o", "u", "g", "l", "a", "s"  
t[3] == "g"  
t[1:3] == ("o", "u")
```
- tuples cannot be changed, but they can be concatenated
- Example:

```
u = ("d",) + t[1:]
```

# Tuple Assignment

- remember, how to exchange two values:
  - Solution 1 (new variable):  $z = y; y = x; x = z$
  - Solution 2 (parallel assign.):  $x, y = y, x$
- now, we see that this is a tuple assignment
- assignment to a tuple is assignment to each tuple element
- works not only with other tuple, but with any sequence
- Example:
  - $x, y, z = [23, 42, -3.0]$
  - $name = \text{"Peter Schneider-Kamp"}$
  - $first, last = name.split()$

# Tuples as Return Values

- useful to return more than one value in a function
- but functions only return one value
- tuples can be used to contain multiple values
- Example 1: built-in function `divmod(x,y)`

```
t = divmod(10, 3)
print t
quot, rem = divmod(101, 17)
```
- Example 2: extract username, hostname, and domain  
`def decompose(email):`

```
    username, rest = email.split("@")
    rest = rest.split(".")
    return username, rest[0], ".".join(rest[1:])
```

# Variable-Length Argument Tuples

- functions can take a variable number of arguments
- arguments are passed as one tuple (*gather*)
- Example 1: function that works similar to `print` statement

```
def printf(*args):      # * indicates variable arguments
    for arg in args:   # iterates through tuple
        print arg,    # prints one argument
    print              # prints new line
```

- Example 2: prints all arguments `n` times

```
def printn(n, *args):
    for arg in args * n:
        print arg
```

# Tuples instead of Arguments

- tuples cannot directly be used instead for normal parameters
- Example:

```
t = (42, 23)
```

```
print divmod(t)      # gives TypeError
```

- using “\*” we can declare that a tuple should be *scattered*
- Example:

```
print divmod(*t)     # prints (1, 19)
```

# Lists and Tuples

- built-in function `zip()` combines two sequences

- Example 1:

```
zip([1, 2, 3], ["c", "b", "a"]) == [(1, "c"), (2, "b"), (3, "a")]
```

- Example 2:

```
zip("You", "suck!") == [("Y", "s"), ("o", "u"), ("u", "c")]
```

- iteration through list of tuples using tuple assignment

- Example:

```
t = [(1, "c"), (2, "b"), (3, "a")]
```

```
for num, ch in t:
```

```
    print "we have paired", num, "and", ch
```

# Lists and Tuples

- with `zip()`, `for` loop, and tuple assignment we can iterate through two sequences in parallel
- Example 1: sum of product of elements (*dot product*)

```
def dot_product(x, y):
```

```
    res = 0
```

```
    for a, b in zip(x, y):
```

```
        res += a*b
```

```
    return res
```

```
dot_product([1, 4, 3], [4, 5, 6])
```

- Example 2: the same shorter ...

```
def dot_product(x, y):
```

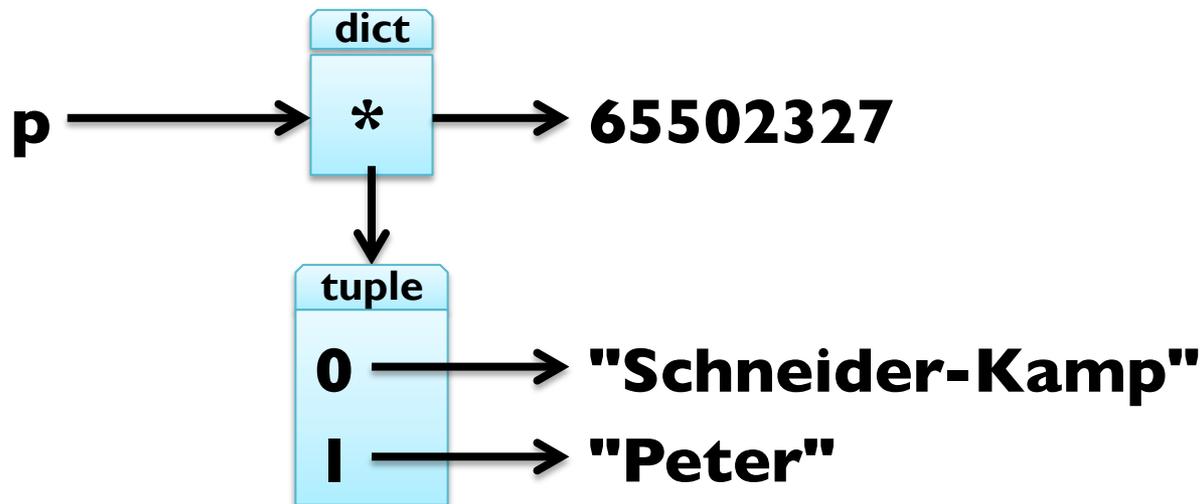
```
    return reduce(lambda x, y: x + y[0] * y[1], zip(x, y), 0)
```

# Dictionaries and Tuples

- dictionaries return a list of tuples with the `items()` method
- Example: `d = {"a" : 3, "b" : 2, "c" : 1}`  
`d.items() == [("a", 3), ("c", 1), ("b", 2)]`
- tuples can also be used to create new dictionary using `dict()`
- Example: `t = [("a", 3), ("c", 1), ("b", 2)]`  
`dict(t) == {"a" : 3, "b" : 2, "c" : 1}`
- combine with `zip()` for easy dictionary generation
- Example: `d = dict(zip("abcdefg", range(7,0,-1)))`
- with tuple assignment and `for` loop, easy traversal
- Example: `for key, val in d.items():`      `print key, val`

# Dictionaries and Tuples

- tuples can be used as dictionary keys (they are immutable)
- Example: `p = {}`; `first = "Peter"`; `last = "Schneider-Kamp"`  
`p[last, first] = 65502327`
- traversal by `for` loop and tuple assignment
- Example 1: `for last, first in p: print first, last, p[last, first]`
- Example 2: `for (last, first), num in p.items(): print last, first, num`



# Dictionaries and Tuples

- tuples can be used as dictionary keys (they are immutable)
- Example: `p = {}`; `first = "Peter"`; `last = "Schneider-Kamp"`  
`p[last, first] = 65502327`
- traversal by `for` loop and tuple assignment
- Example 1: `for last, first in p: print first, last, p[last, first]`
- Example 2: `for (last, first), num in p: print last, first, num`



# Comparing Tuples

- comparing tuples same as comparing any sequence
- like with strings, sequences are compared lexicographically
- Example:  $(3,) > (2, 2, 2)$   
 $(1, 2, 3, 4, 5) < (1, 2, 3, 5, 5)$
- tuples can be used to sort lists after arbitrary criteria
- Example: sort list of words after their length, shortest last

```
def sort_by_length(words):
```

```
    t = []; res = []
```

```
    for word in words:          t.append((len(word), word))
```

```
    t.sort(reverse=True)
```

```
    for length, word in t:      res.append(word)
```

```
    return res
```

# Comparing Tuples

- comparing tuples same as comparing any sequence
- like with strings, sequences are compared lexicographically
- Example:  $(3,) > (2, 2, 2)$   
 $(1, 2, 3, 4, 5) < (1, 2, 3, 5, 5)$
- tuples can be used to sort lists after arbitrary criteria
- Example: sort list of words after their length, shortest last

```
def sort_by_length(words):
```

```
    t = map(lambda x: (len(x), x), words)
```

```
    t.sort(reverse=True)
```

```
    return map(lambda pair: pair[1], t)
```

# Sequences of Sequences

- most sequences can contain other types of sequences
- string is an exception, as it only contains characters
- can always use a list of characters instead of string
- lists usually preferred to tuples (they are mutable)
- in some situations, tuples more often used:
  1. tuples are more “easy” to construct, e.g. `return n, n**2`
  2. tuples can be dictionary keys (they are immutable)
  3. tuples are safer due to “aliasing”, so use them e.g. as sequence arguments to functions
- methods `sort()` and `reverse()` not available for tuples
- use functions `sorted(iterable)` and `reversed(iterable)` instead

# Debugging Shape Errors

- lists, dictionaries, and tuples are *data structures*
- combining this, we obtain compound data structures
- this gives rise to new errors, so called **shape errors**
- a shape error is when the structure of the compound data structure does not fit its use
- Example: 

```
d = {"Schneider-Kamp", "Peter"} : 65502327
```

```
for last, first, number in d: print number
```
- use **structshape** module for debugging
- available from <http://thinkpython.com/code/structshape.py>
- Example: 

```
from structshape import structshape
```

```
structshape(d) == "dict of 1 tuple of 2 str->int"
```

# **SELECTING DATA STRUCTURES**

# Reading and Cleaning Words

1. read file given as argument
  2. break lines into words
  3. strip whitespace & punctuation
  4. convert to lower-case letters
- import module `sys` for command line arguments `sys.argv`
  - Example: `import sys; print sys.argv`
  - import module `string` for punctuation
  - Example: `import string; print string.punctuation`
  - use `translate(None, deletechars)` to remove punctuation
  - Example: `"Hello World!".translate(None, "o!")`