Containers

Announcements

Iteration and Recursion

Replacing Iteration with Recursion

Replacing Iteration with Recursion

You can convert while/for statements to recursion without inventing new logic:

Replacing Iteration with Recursion

You can convert while/for statements to recursion without inventing new logic:

- Each pass through the body of a while/for statement is replaced by a recursive call.

Replacing Iteration with Recursion

You can convert while/for statements to recursion without inventing new logic:

- Each pass through the body of a while/for statement is replaced by a recursive call.
- Instead of using assignment statements, assign names to values using a call expression.

Replacing Iteration with Recursion

You can convert while/for statements to recursion without inventing new logic:

- Each pass through the body of a while/for statement is replaced by a recursive call.
- Instead of using assignment statements, assign names to values using a call expression.
- If needed, introduce a new function with an argument for every value that must be tracked.

Replacing Iteration with Recursion

You can convert while/for statements to recursion without inventing new logic:

- Each pass through the body of a while/for statement is replaced by a recursive call.
- Instead of using assignment statements, assign names to values using a call expression.
- If needed, introduce a new function with an argument for every value that must be tracked.

(Demo)

Box-and-Pointer Notation

The Closure Property of Data Types

The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:

  The result of combination can itself be combined using the same method

The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:

  The result of combination can itself be combined using the same method

- Closure is powerful because it permits us to create hierarchical structures

The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:

  The result of combination can itself be combined using the same method

- Closure is powerful because it permits us to create hierarchical structures

- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:

  The result of combination can itself be combined using the same method

- Closure is powerful because it permits us to create hierarchical structures

- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

Lists can contain lists as elements (in addition to anything else)

Box-and-Pointer Notation in Environment Diagrams

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element

Each box either contains a primitive value or points to a compound value

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element

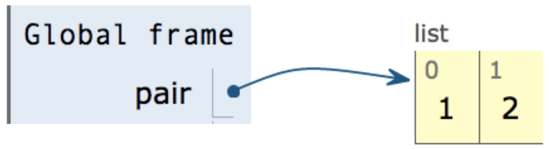
Each box either contains a primitive value or points to a compound value

```
pair = [1, 2]
```

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element

Each box either contains a primitive value or points to a compound value

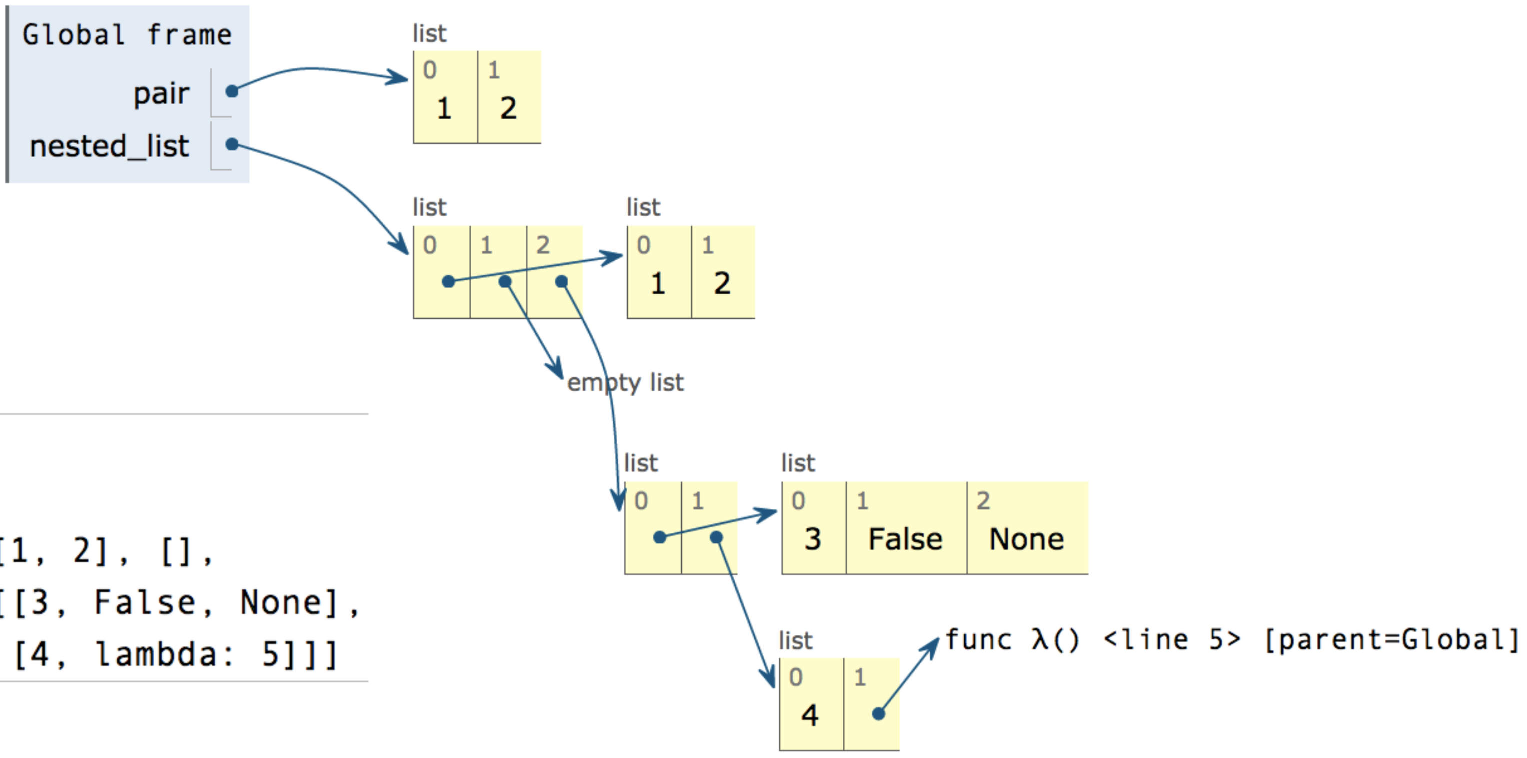


```
pair = [1, 2]
```

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element

Each box either contains a primitive value or points to a compound value

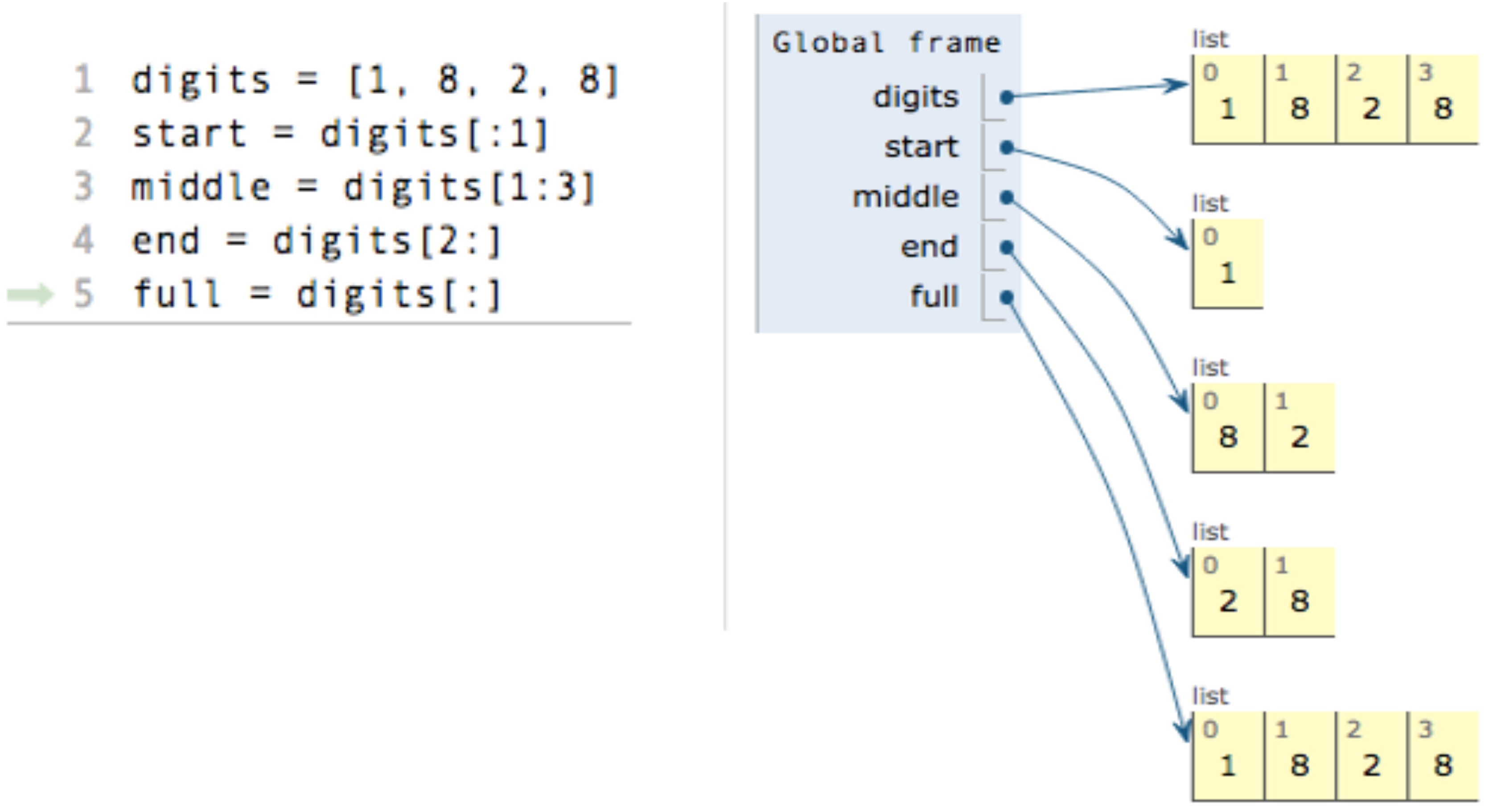


```
1  pair = [1, 2]
2
3  nested_list = [[1, 2], [],
4                 [[3, False, None],
5                  [4, lambda: 5]]]
```

pythontutor.com/composingprograms.html#code=pair%20%3D%20[1,%202]%0A%0Anested_list%20%3D%20[[1,%202],%20[],%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20[[3,%20False,%20None],%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20[4,%20lambda%3A%205]]]&mode=display&origin=composingprograms.js&cumulative=true&py=3&rawInputLstJSON=[]&curInstr=4

Slicing Creates Lists

Slicing Creates New Values



```
digits = [1, 8, 2, 8]
start = digits[:1]
middle = digits[1:3]
end = digits[2:]
full = digits[:]
```

pythontutor.com/composingprograms.html#code=digits%20%3D%20[1,%208,%202,%208]%0Astart%20%3D%20digits[%3A1]%0Amiddle%20%3D%20digits[1%3A3]%0Aend%20%3D%20digits[2%3A]%0Afull%20%3D%20digits[%3A]&cumulative%3Dtrue&curInstr%3D5&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=[]

Processing Container Values

(Demo)

Aggregation

Aggregation

```
Several built-in functions take iterable arguments and aggregate them into a value
```

Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- **sum**(iterable[, start]) -> value

  ```
  Return the sum of an iterable (not of strings) plus the value
  of parameter 'start' (which defaults to 0).  When the iterable is
  empty, return start.
  ```

Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- **sum**(iterable[, start]) -> value

  ```
  Return the sum of an iterable (not of strings) plus the value
  of parameter 'start' (which defaults to 0).  When the iterable is
  empty, return start.
  ```

- **max**(iterable[, key=func]) -> value
  max(a, b, c, ...[, key=func]) -> value

  ```
  With a single iterable argument, return its largest item.
  With two or more arguments, return the largest argument.
  ```

Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- **sum**(iterable[, start]) -> value

  ```
  Return the sum of an iterable (not of strings) plus the value
  of parameter 'start' (which defaults to 0).  When the iterable is
  empty, return start.
  ```

- **max**(iterable[, key=func]) -> value
  max(a, b, c, ...[, key=func]) -> value

  ```
  With a single iterable argument, return its largest item.
  With two or more arguments, return the largest argument.
  ```

- **all**(iterable) -> bool

  ```
  Return True if bool(x) is True for all values x in the iterable.
  If the iterable is empty, return True.
  ```

Discussion Question

```
Find the power of 2 that is closest to 1,000 using one line:

(You can assume that it's smaller than 2 ** 100.)
```

```
min( [ ______________________________ ], key= _________________________ )
```

Discussion Question

```
Find the power of 2 that is closest to 1,000 using one line:

(You can assume that it's smaller than 2 ** 100.)
```

```
          2 ** n for n in range(100)
min( [ ____________________________ ], key= _______________________ )
```

Discussion Question

Find the power of 2 that is closest to 1,000 using one line:

(You can assume that it's smaller than 2 ** 100.)

```
min( [ 2 ** n for n in range(100) ], key= lambda x: abs(1000 - x) )
```

Strings

Strings are an Abstraction

Strings are an Abstraction

Representing data:

```
'200'      '1.2e-5'      'False'      '[1, 2]'
```

Strings are an Abstraction

Representing data:

```
'200'      '1.2e-5'      'False'      '[1, 2]'
```

Representing language:

```
"""And, as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.
"""
```

Strings are an Abstraction

Representing data:

```
'200'      '1.2e-5'       'False'       '[1, 2]'
```

Representing language:

```
"""And, as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.
"""
```

Representing programs:

```
'curry = lambda f: lambda x: lambda y: f(x, y)'
```

Strings are an Abstraction

Representing data:

```
'200'      '1.2e-5'      'False'      '[1, 2]'
```

Representing language:

```
"""And, as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.
"""
```

Representing programs:

```
'curry = lambda f: lambda x: lambda y: f(x, y)'
```

(Demo)

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'
```

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'
```

Single-quoted and double-quoted strings are equivalent

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'


>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
```

Single-quoted and double-quoted strings are equivalent

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'


>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
```

Single-quoted and double-quoted strings are equivalent

A backslash "escapes" the following character

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'


>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
```

Single-quoted and double-quoted strings are equivalent

A backslash "escapes" the following character

"Line feed" character represents a new line

Dictionaries

```
{'Dem': 0}
```

Limitations on Dictionaries

Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)

Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)
- Two **keys cannot be equal;** There can be at most one value for a given key

Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)
- Two **keys cannot be equal;** There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)

- Two **keys cannot be equal;** There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)
- Two **keys cannot be equal;** There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

If you want to associate multiple values with a key, store them all in a sequence value

Dictionary Comprehensions

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}

Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

```
Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

An expression that evaluates to a dictionary using this evaluation procedure:

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

Short version: `{<key exp>: <value exp> for <name> in <iter exp>}`

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}

Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty *result dictionary* that is the value of the expression

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

Short version: `{<key exp>: <value exp> for <name> in <iter exp>}`

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent

2. Create an empty *result dictionary* that is the value of the expression

3. For each element in the iterable value of `<iter exp>`:

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}

Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty *result dictionary* that is the value of the expression
3. For each element in the iterable value of <iter exp>:
   A. Bind <name> to that element in the new frame from step 1

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}

Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty *result dictionary* that is the value of the expression
3. For each element in the iterable value of `<iter exp>`:
   - A. Bind `<name>` to that element in the new frame from step 1
   - B. If `<filter exp>` evaluates to a true value, then add to the result dictionary an entry that pairs the value of `<key exp>` to the value of `<value exp>`

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}

Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty *result dictionary* that is the value of the expression
3. For each element in the iterable value of `<iter exp>`:
   A. Bind `<name>` to that element in the new frame from step 1
   B. If `<filter exp>` evaluates to a true value, then add to the result dictionary an entry that pairs the value of `<key exp>` to the value of `<value exp>`

`{x * x: x for x in [1, 2, 3, 4, 5] if x > 2}` evaluates to `{9: 3, 16: 4, 25: 5}`

Example: Indexing

Implement **index**, which takes a sequence of **keys**, a sequence of **values**, and a two-argument **match** function. It returns a dictionary from **keys** to lists in which the list for a key k contains all **values** v for which **match**(k, v) is a true value.

```
def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which
    match(k, v) is a true value.

    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """

    return ______________________________________________________
```

Example: Indexing

Implement **index**, which takes a sequence of **keys**, a sequence of **values**, and a two-argument **match** function. It returns a dictionary from **keys** to lists in which the list for a key k contains all **values** v for which **match**(k, v) is a true value.

```
def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which
    match(k, v) is a true value.

    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """
    return  {k:                                         for k in keys}
           ___________________________________________________________
```

Example: Indexing

Implement **index**, which takes a sequence of **keys**, a sequence of **values**, and a two-argument **match** function. It returns a dictionary from **keys** to lists in which the list for a key k contains all **values** v for which **match**(k, v) is a true value.

```
def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which
    match(k, v) is a true value.

    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """
    return {k: [v for v in values if match(k, v)] for k in keys}
```