Example

list

List of squares

Program to generate a list containing squares of `n` integers starting from `0`

```
n = 12

squares = []
for i in range(n):
    squares.append(i**2)
print(squares)
```

```
$ python3 squares.py
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Example

List comprehension

- The previous construct is common

  So Python provides a shorter syntactic equivalent

```
n = 12
squares = [i**2 for i in range(n)]
```

```
$ python3 squares.py
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

- Inside square brackets

  An expression followed by a for clause followed by zero or more for or if clauses

Mapping Type

Dictionary `dict`

- Lists and tuples can only be indexed with integers
- Dictionaries can be indexed with almost any (immutable) type including strings, tuples and integers
- Dictionaries can be thought of as key, value pairs: The key is the index used to retrieve a value

Creating dictionaries

- Dictionaries created by enclosing comma separated key, value pairs in braces
  - Each key and value separated by a colon :

```
>>> mydict = {'one':1, 'two':2, 'tuple':(1,2)}
>>> type(mydict)
<class 'dict'>
```

- Can also be created from a list of 2-tuples

```
>>> alst = [('one',1), ('two',2), ('three',3)]
>>> amap = dict(alst)
>>> print(amap)
{'one': 1, 'two': 2, 'three': 3}
```

Indexing dictionaries

- Key enclosed in square brackets

```
>>> mydict = {'one':1, 'two':2, 'tuple':(1,2)}
>>> print(mydict['two'])
2
>>> print(mydict['tuple'])
(1, 2)
```

- Invalid key raises an exception

```
>>>> print(mydict['list'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'list'
```

Adding items to dictionaries

```
>>>> adict = {}
>>>> print(adict)
{}
>>>> adict['answertolife'] = 42
>>>> adict[(2,3)] = '2 comma 3'

>>>> print(adict)
{'answertolife': 42, (2, 3): '2 comma 3'}
```

```
>>>> del adict[(2,3)]  #also works for lists
>>>> print(adict)
{'answertolife': 42}
```

Iterator Protocol

- Dictionaries can be used as iterators in `for` loops
- Iterator returns keys
- dict is unordered

```
>>> adict = {'name':'John', 'city':'Philadelphia', 'employer':'Temple'}
>>> for key in adict:
...     print(key)
...
city
name
employer
```

Python Types

Dictionary Methods

Some methods

- `items()`
- `values()`

  See `help(dict)`

```
>>> adict = {'name':'John', 'city':'Philadelphia', 'employer':'Temple'}

>>> for item in adict.items():
...     print(item)
...
('city', 'Philadelphia')
('name', 'John')
('employer', 'Temple')

>>> for value in adict.values():
...     print(value)
...
Philadelphia
John
Temple
```

Membership Testing

- `in` tests for the existence of a key in the `dict`

```
>>>> adict = {'name':'John', 'city':'Philadelphia', 'employer':'Temple'}
>>>> 'name' in adict
True
>>>> 'Temple' in adict
False
```

- Unlike lists, membership testing in dictionaries takes constant time

Example

Program to count number of occurrences of each character in a string

Implementation

Example

```
hist = {}
for char in instr.lower():
     if char.isalpha():
          if char not in hist:
               hist[char] = 1
          else:
               hist[char] += 1

for char, num in hist.items():
     print(char,': ', '+'*num)
```

Functions

Functions

- Above code needs to be copied wherever it is needed
  - Variable name needs to be adjusted
  - Leads to duplicated code
  - Makes program less readable
- Functions allow us to package code so that it is reusable
  - Code can be implemented and tested more independently of the rest of the program
  - Keeping interface fixed gives programmer freedom to modify implementation
    - While avoiding changes to the rest of the program
  - Functions can return an object. Default is the object `None`

Functions

Implementation

As a function

```
def charcount(s):
     hist = {}
     for char in s.lower():
          if char.isalpha():
               if char not in hist:
                    hist[char] = 1
               else:
                    hist[char] += 1
     return hist

def printhist(d):
     for char, num in d.items():
          print(char,': ', '+'*num)
```

Parameters and Arguments

Functions

Namespaces

```
def charcount(s):
     hist = {}
     for char in s.lower():
          if char.isalpha():
               if char not in hist:
                    hist[char] = 1
               else:
                    hist[char] += 1
     return hist

mydict = charhisto('hello')
```

- In the function definition `'s'` is a parameter to `charhisto`
- When the function is called, an argument is passed
  - Above, the argument is the string `'hello'`
- This argument is assigned to the parameter `'s'`
  - `s = 'hello'` before the body of the function is executed

Parameters and Arguments

- Functions can define multiple parameters
- Arguments are assigned to the parameters based on their position
  - First argument always assigned to `param1`

```
>>>> def myfunc(param1, param2, param3):
....            print('first argument is', param1)
....            print('second argument is', param2)
....            print('third argument is', param3)
....
>>>> myfunc(1,2,3)
first argument is 1
second argument is 2
third argument is 3

>>>> myfunc(3,2,1)
first argument is 3
second argument is 2
third argument is 1
```

Keyword arguments

- Functions can be alternatively called with `parameter = argument` syntax
- Argument is assigned to the named parameter

```
>>>> def myfunc(param1, param2, param3):
....          print('first argument is', param1)
....          print('second argument is', param2)
....          print('third argument is', param3)
....
>>>> myfunc(param3 = 3, param1 = 1, param2 = 2)
first argument is 1
second argument is 2
third argument is 3

>>>> myfunc(param3 = 1, param2 = 2, param1 = 3)
first argument is 3
second argument is 2
third argument is 1
```

Default Arguments

- Allow us to specify default values for a parameter if no argument is passed

```
>>>> def createlist(nelem, value):
....          return [value]*nelem
....
>>>> createlist(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: createlist() missing 1 required positional argument: 'value'

>>>> def createlist(nelem, value=None):
....      return [value]*nelem
....
>>>> createlist(5)
[None, None, None, None, None]

>>>> createlist(5,5)
[5, 5, 5, 5, 5]
```

Python

Local variables

- Variable names inside a function can only be seen within that function
  - They are local to that function
  - And independent of any variable names existing outside the function
    - Even ones with the same name
- That is creating a function defines a new *namespace*
- Namespace is the table that keeps track of variable names and the objects they reference

Variable Scope


```
def charcount(s):
     hist = {}
     for char in s.lower():
          if char.isalpha():
               if char not in hist:
                    hist[char] = 1
               else:
                    hist[char] += 1
     return hist
```

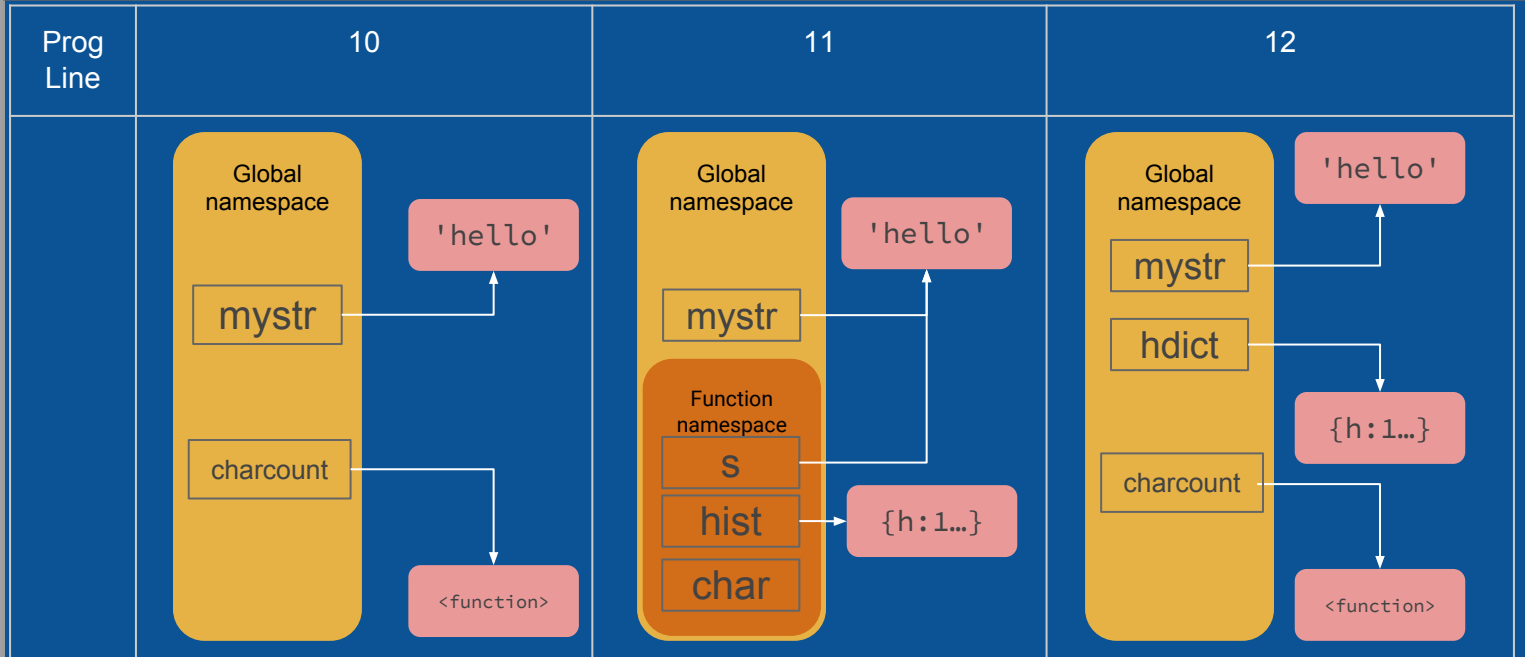
- Variables `hist` and `s` are local to the function `charhisto()`
  - When `charhisto()` is called, a new namespace is defined
  - `hist` and `s` are variable names defined in that new namespace
  - Names `hist` and `s` cease to exist after function ends/returns

Functions

Namespaces

```
1   def charcount(s):
        ...
        # omitted for brevity
        ...
9       return hist

10  mystr = 'hello'
11  hdict = charcount(mystr)
12  print(hdict)
```



Python

`locals()`, `globals()`

- Built-in functions that return names of variables in local scope and global scope respectively

Mutable function arguments

- Functions can modify mutable arguments, like lists and dictionaries in-place

```
>>>> def modval(l):
....          l[0] = 9999
....

>>>> alst = [100, 200, 300]
>>>> modval(alst)
>>>> print(alst)
[9999, 200, 300]
```

Modules

Modules

- Python has no built-in function for returning the square root
- Function to calculate (real) quadratic roots
- Python has a (standard) library of mathematical functions and constants called `math`
- `import` creates new namespace `math`

```
import math

def quadroot(a, b, c):
    discr = (b * b) - (4 * a * c)
    discr_sqrt = math.sqrt(discr)
    root1 = (-b + discr_sqrt) / (2 * a)
    root2 = (-b - discr_sqrt) / (2 * a)
    return root1, root2

r1, r2 = quadroot(1, 3, -10)
print r1, r2
```

Creating modules

- We can import quadroot.py in another Python program
- `import quadroot`
  - Searches for `quadroot.py` in
    Built-in modules
    Directory containing the Python program
    `PYTHONPATH`

```
import quadroot

a, b = quadroot.quadroot(5, 3, -10)
print(a, b)
```

```
$ python3 quadprog.py
2.0 -5.0
1.145683229480096 -1.7456832294800961
```

Creating modules

- `import` causes statements in the module to be executed
- Global variable `__name__` contain name of module when imported and `'__main__'` when executed as a script/program

```
import math

def quadroot(a, b, c):
    discr = (b * b) - (4 * a * c)
    discr_sqrt = math.sqrt(discr)
    root1 = (-b + discr_sqrt) / (2 * a)
    root2 = (-b - discr_sqrt) / (2 * a)
    return root1, root2

if __name__ == '__main__':
    r1, r2 = quadroot(1, 3, -10)
    print(r1, r2)
```