Procedural Programming: Defining Functions

Introduction to Programming in Python

Outline

Outline

Outline

Outline

Outline

Outline

Function Definition

Function Definition

A function is a callable unit of code with a well-defined interface and behavior

Function Definition

A function is a callable unit of code with a well-defined interface and behavior

Function definition:

```
def <name>(<param1>, <param2>, ...):
    <stmt1>
    <stmt2>
    ...
```

Function Definition

Function Definition ▶ Return Statement

A return statement is used to transfer control back to the caller, sometimes with a value

Function Definition ▶ Return Statement

A return statement is used to transfer control back to the caller, sometimes with a value

Return statement:

```
return
return <exp>
```

Function Definition

Function Definition ▶ Control Flow

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

`__name__`	
`main::x`	
`main::y`	
`_square::x`	
`_square::y`	

```
$ _
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

__name__	"__main__"
main::x	
main::y	
_square::x	
_square::y	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

__name__	"__main__"
main::x	
main::y	
_square::x	
_square::y	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

__name__	"__main__"
main::x	
main::y	
_square::x	
_square::y	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

__name__	"__main__"
main::x	
main::y	
_square::x	
_square::y	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

__name__	"__main__"
main::x	
main::y	
_square::x	
_square::y	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```python
import stdio
import sys

def main():
    x = int(sys.argv[1])
    y = _square(x)
    stdio.writeln(y)

def _square(x):
    y = x * x
    return y

if __name__ == "__main__":
    main()
```

__name__	"__main__"
main::x	
main::y	
_square::x	
_square::y	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

__name__	"__main__"
main::x	
main::y	
_square::x	
_square::y	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

`__name__`	`"__main__"`
`main::x`	`11`
`main::y`	
`_square::x`	
`_square::y`	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

`__name__`	`"__main__"`
`main::x`	`11`
`main::y`	
`_square::x`	
`_square::y`	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

`__name__`	`"__main__"`
`main::x`	11
`main::y`	
`_square::x`	11
`_square::y`	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

`__name__`	`"__main__"`
`main::x`	11
`main::y`	
`_square::x`	11
`_square::y`	121

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```python
import stdio
import sys

def main():
    x = int(sys.argv[1])
    y = _square(x)
    stdio.writeln(y)

def _square(x):
    y = x * x
    return y

if __name__ == "__main__":
    main()
```

`__name__`	`"__main__"`
`main::x`	`11`
`main::y`	
`_square::x`	`11`
`_square::y`	`121`

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

`__name__`	`"__main__"`
`main::x`	`11`
`main::y`	`121`
`_square::x`	
`_square::y`	

```
$ python3 square.py 11
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

`__name__`	`"__main__"`
`main::x`	`11`
`main::y`	`121`
`_square::x`	
`_square::y`	

```
$ python3 square.py 11
121
```

Function Definition ▶ Control Flow

Function Call Trace (`square.py`):

```
 1 import stdio
 2 import sys
 3
 4 def main():
 5     x = int(sys.argv[1])
 6     y = _square(x)
 7     stdio.writeln(y)
 8
 9 def _square(x):
10     y = x * x
11     return y
12
13 if __name__ == "__main__":
14     main()
```

`__name__`	
`main::x`	
`main::y`	
`_square::x`	
`_square::y`	

```
$ python3 square.py 11
121
$ _
```

Function Definition

Function Definition ▶ Salient Features

Inputs to a function are generally its parameters

Function Definition ▶ Salient Features

Inputs to a function are generally its parameters

A function generally does not write any output — its output is its return value

Function Definition ▶ Salient Features

Inputs to a function are generally its parameters

A function generally does not write any output — its output is its return value

A function may have side effects

Function Definition ▶ Salient Features

Inputs to a function are generally its parameters

A function generally does not write any output — its output is its return value

A function may have side effects

Arguments to a function are passed by value

Function Definition ▶ Salient Features

Inputs to a function are generally its parameters

A function generally does not write any output — its output is its return value

A function may have side effects

Arguments to a function are passed by value

A non-void function can only return a single value but may have multiple return statements

Function Definition ▶ Salient Features

Inputs to a function are generally its parameters

A function generally does not write any output — its output is its return value

A function may have side effects

Arguments to a function are passed by value

A non-void function can only return a single value but may have multiple return statements

The scope of a function's parameters and other variables is limited to that function

Function Definition

Function Definition ▶ Default Arguments

Function Definition ▶ Default Arguments

A function may designate an argument to be optional by specifying a default value for that argument

Function Definition ▶ Default Arguments

A function may designate an argument to be optional by specifying a default value for that argument

Example (computing $H_{n,r} = 1 + 1/2^r + 1/3^r + \cdots + 1/n^r$):

Function Definition ▶ Default Arguments

A function may designate an argument to be optional by specifying a default value for that argument

Example (computing $H_{n,r} = 1 + 1/2^r + 1/3^r + \cdots + 1/n^r$):

```
def harmonic(n, r = 1):
    total = 0.0
    for i in range(1, n + 1):
        total += 1 / (i ** r)
    return total
```

Function Definition ▶ Default Arguments

A function may designate an argument to be optional by specifying a default value for that argument

Example (computing $H_{n,r} = 1 + 1/2^r + 1/3^r + \cdots + 1/n^r$):

```
def harmonic(n, r = 1):
    total = 0.0
    for i in range(1, n + 1):
        total += 1 / (i ** r)
    return total
```

Calling `harmonic(5)` is the same as calling `harmonic(5, 1)`

Function Definition

Function Definition ▶ Mutable Arguments

Function Definition ▶ Mutable Arguments

Changing a parameter that refers to a mutable object also changes the object's value in the calling code

Function Definition ▶ Mutable Arguments

Changing a parameter that refers to a mutable object also changes the object's value in the calling code

Example:

Function Definition ▶ Mutable Arguments

Changing a parameter that refers to a mutable object also changes the object's value in the calling code

Example:

```
import stdio

def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp

a = [1, 2, 3, 4, 5]
exchange(a, 1, 3)
stdio.writeln(a)
```

Function Definition ▶ Mutable Arguments

Changing a parameter that refers to a mutable object also changes the object's value in the calling code

Example:

```
import stdio

def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp

a = [1, 2, 3, 4, 5]
exchange(a, 1, 3)
stdio.writeln(a)
```

```
[1, 4, 3, 2, 5]
```

Examples

Examples

Program (`harmonicredux.py`):

Examples

Program (`harmonicredux.py`):

- Command-line input: `n` (int)

Examples

Program (`harmonicredux.py`):

- Command-line input: `n` (int)
- Standard output: the `n`th harmonic number, $H_{\texttt{n}} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\texttt{n}} \approx \ln(\texttt{n}) + 0.57721$

Examples

Program (`harmonicredux.py`):

- Command-line input: `n` (int)
- Standard output: the `n`th harmonic number, $H_{\texttt{n}} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\texttt{n}} \approx \ln(\texttt{n}) + 0.57721$

```
$ _
```

Examples

Program (`harmonicredux.py`):

- Command-line input: `n` (int)
- Standard output: the `n`th harmonic number, $H_{\texttt{n}} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\texttt{n}} \approx \ln(\texttt{n}) + 0.57721$

```
$ python3 harmonicredux.py 10
```

Examples

Program (`harmonicredux.py`):

- Command-line input: `n` (int)
- Standard output: the `n`th harmonic number, $H_{\texttt{n}} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\texttt{n}} \approx \ln(\texttt{n}) + 0.57721$

```
$ python3 harmonicredux.py 10
2.9289682539682538
$ _
```

Examples

Program (`harmonicredux.py`):

- Command-line input: `n` (int)
- Standard output: the `n`th harmonic number, $H_{\texttt{n}} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\texttt{n}} \approx \ln(\texttt{n}) + 0.57721$

```
$ python3 harmonicredux.py 10
2.9289682539682538
$ python3 harmonicredux.py 1000
```

Examples

Program (`harmonicredux.py`):

- Command-line input: `n` (int)
- Standard output: the `n`th harmonic number, $H_{\texttt{n}} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\texttt{n}} \approx \ln(\texttt{n}) + 0.57721$

```
$ python3 harmonicredux.py 10
2.9289682539682538
$ python3 harmonicredux.py 1000
7.485470860550343
$ _
```

Examples

Program (`harmonicredux.py`):

- Command-line input: `n` (int)
- Standard output: the `n`th harmonic number, $H_{\texttt{n}} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\texttt{n}} \approx \ln(\texttt{n}) + 0.57721$

```
$ python3 harmonicredux.py 10
2.9289682539682538
$ python3 harmonicredux.py 1000
7.485470860550343
$ python3 harmonicredux.py 10000
```

Examples

Program (`harmonicredux.py`):

- Command-line input: `n` (int)
- Standard output: the `n`th harmonic number, $H_{\texttt{n}} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\texttt{n}} \approx \ln(\texttt{n}) + 0.57721$

```
$ python3 harmonicredux.py 10
2.9289682539682538
$ python3 harmonicredux.py 1000
7.485470860550343
$ python3 harmonicredux.py 10000
9.787606036044348
$ _
```

Examples

Examples

```
import stdio
import sys

def main():
    n = int(sys.argv[1])
    stdio.writeln(_harmonic(n))

def _harmonic(n):
    total = 0.0
    for i in range(1, n + 1):
        total += 1 / i
    return total

if __name__ == "__main__":
    main()
```

Examples

Examples

Program (`couponcollectorredux.py`):

Examples

Program (`couponcollectorredux.py`):

- Command-line input: `n` (int)

Examples

Program (`couponcollectorredux.py`):

- Command-line input: `n` (int)
- Standard output: number of coupons collected before obtaining at least one of `n` unique coupons

Examples

Program (`couponcollectorredux.py`):

- Command-line input: `n` (int)
- Standard output: number of coupons collected before obtaining at least one of `n` unique coupons

```
$ _
```

Examples

Program (`couponcollectorredux.py`):

- Command-line input: `n` (int)
- Standard output: number of coupons collected before obtaining at least one of `n` unique coupons

```
$ python3 couponcollectorredux.py 1000
```

Examples

Program (`couponcollectorredux.py`):

- Command-line input: `n` (int)
- Standard output: number of coupons collected before obtaining at least one of `n` unique coupons

```
$ python3 couponcollectorredux.py 1000
7173
$ _
```

Examples

Program (`couponcollectorredux.py`):

- Command-line input: `n` (int)
- Standard output: number of coupons collected before obtaining at least one of `n` unique coupons

```
$ python3 couponcollectorredux.py 1000
7173
$ python3 couponcollectorredux.py 10000
```

Examples

Program (`couponcollectorredux.py`):

- Command-line input: `n` (int)
- Standard output: number of coupons collected before obtaining at least one of `n` unique coupons

```
$ python3 couponcollectorredux.py 1000
7173
$ python3 couponcollectorredux.py 10000
94718
$ _
```

Examples

Program (`couponcollectorredux.py`):

- Command-line input: `n` (int)
- Standard output: number of coupons collected before obtaining at least one of `n` unique coupons

```
$ python3 couponcollectorredux.py 1000
7173
$ python3 couponcollectorredux.py 10000
94718
$ python3 couponcollectorredux.py 100000
```

Examples

Program (`couponcollectorredux.py`):

- Command-line input: `n` (int)
- Standard output: number of coupons collected before obtaining at least one of `n` unique coupons

```
$ python3 couponcollectorredux.py 1000
7173
$ python3 couponcollectorredux.py 10000
94718
$ python3 couponcollectorredux.py 100000
1110981
$ _
```

Examples

Examples

```
import stdarray
import stdio
import stdrandom
import sys

def main():
    n = int(sys.argv[1])
    stdio.writeln(_collect(n))

def _collect(n):
    count = 0
    collectedCount = 0
    isCollected = stdarray.create1D(n, False)
    while collectedCount < n:
        value = _getCoupon(n)
        count += 1
        if not isCollected[value]:
            collectedCount += 1
            isCollected[value] = True
    return count

def _getCoupon(n):
    return stdrandom.uniformInt(0, n)

if __name__ == "__main__":
    main()
```

Examples

Examples

Program (`playthattunedeluxe.py`):

Examples

Program (`playthattunedeluxe.py`):

- Standard input: sound samples, each characterized by a pitch and a duration

Examples

Program (`playthattunedeluxe.py`):

- Standard input: sound samples, each characterized by a pitch and a duration
- Standard audio output: the tune

Examples

Program (`playthattunedeluxe.py`):

- Standard input: sound samples, each characterized by a pitch and a duration
- Standard audio output: the tune

```
$ _
```

Examples

Program (`playthattunedeluxe.py`):

- Standard input: sound samples, each characterized by a pitch and a duration
- Standard audio output: the tune

```
$ cat data/looney.txt
```

Examples

Program (`playthattunedeluxe.py`):

- Standard input: sound samples, each characterized by a pitch and a duration
- Standard audio output: the tune

```
$ cat data/looney.txt
7 .270
5 .090
3 .180
...
10 .720
$ _
```

Examples

Program (`playthattunedeluxe.py`):

- Standard input: sound samples, each characterized by a pitch and a duration
- Standard audio output: the tune

```
$ cat data/looney.txt
7 .270
5 .090
3 .180
...
10 .720
$ python3 playthattunedeluxe.py < data/looney.txt
```

Examples

Program (`playthattunedeluxe.py`):

- Standard input: sound samples, each characterized by a pitch and a duration
- Standard audio output: the tune

```
$ cat data/looney.txt
7 .270
5 .090
3 .180
...
10 .720
$ python3 playthattunedeluxe.py < data/looney.txt
$ _
```

Examples

Examples

```
import math
import stdarray
import stdaudio
import stdio

def main():
    while not stdio.isEmpty():
        pitch = stdio.readInt()
        duration = stdio.readFloat()
        stdaudio.playSamples(_createRichNote(pitch, duration))
    stdaudio.wait()

def _createRichNote(pitch, duration):
    NOTES_ON_SCALE = 12
    CONCERT_A = 440.0
    hz = CONCERT_A * pow(2, pitch / NOTES_ON_SCALE)
    mid = _createNote(hz, duration)
    hi = _createNote(2 * hz, duration)
    lo = _createNote(hz / 2, duration)
    hiAndLo = _superpose(hi, lo, 0.5, 0.5)
    return _superpose(mid, hiAndLo, 0.5, 0.5)

def _createNote(hz, duration):
    SPS = 44100
    n = int(SPS * duration)
    note = stdarray.create1D(n + 1, 0.0)
```

Examples

Examples

```
27      for i in range(n + 1):
28          note[i] = math.sin(2 * math.pi * i * hz / SPS)
29      return note
30
31  def _superpose(a, b, aWeight, bWeight):
32      c = stdarray.create1D(len(a), 0.0)
33      for i in range(len(a)):
34          c[i] = a[i] * aWeight + b[i] * bWeight
35      return c
36
37  if __name__ == "__main__":
38      main()
```

Special Functions

Special Functions

Functions in Python are first-class objects, meaning:

Special Functions

Functions in Python are first-class objects, meaning:

- They can receive functions as arguments

Special Functions

Functions in Python are first-class objects, meaning:

- They can receive functions as arguments
- They can return functions as results

Special Functions

Special Functions ▶ `filter()` Function

Special Functions ▶ `filter()` Function

`filter(f, seq)` returns those items of `seq` for which `f(item)` is `True`

Special Functions ▶ `filter()` Function

`filter(f, seq)` returns those items of `seq` for which `f(item)` is `True`

Example:

```
```

Special Functions ▶ `filter()` Function

`filter(f, seq)` returns those items of `seq` for which `f(item)` is `True`

Example:

```
>>> _
```

Special Functions ▶ `filter()` Function

`filter(f, seq)` returns those items of `seq` for which `f(item)` is `True`

Example:

```
>>> primes = filter(isPrime, range(11))
```

Special Functions ▶ `filter()` Function

`filter(f, seq)` returns those items of `seq` for which `f(item)` is `True`

Example:

```
>>> primes = filter(isPrime, range(11))
>>> _
```

Special Functions ▶ `filter()` Function

`filter(f, seq)` returns those items of `seq` for which `f(item)` is `True`

Example:

```
>>> primes = filter(isPrime, range(11))
>>> list(primes)
```

Special Functions ▶ `filter()` Function

`filter(f, seq)` returns those items of `seq` for which `f(item)` is `True`

Example:

```
>>> primes = filter(isPrime, range(11))
>>> list(primes)
[2, 3, 5, 7]
>>> _
```

Special Functions

Special Functions ▶ Lambda Function

A lambda function is an anonymous function defined using the `lambda` keyword

Special Functions ▶ Lambda Function

A lambda function is an anonymous function defined using the `lambda` keyword

The function allows us to create small, one-line functions, often used for short tasks

Special Functions ▶ Lambda Function

A lambda function is an anonymous function defined using the `lambda` keyword

The function allows us to create small, one-line functions, often used for short tasks

Example:

```
```

Special Functions ▶ Lambda Function

A lambda function is an anonymous function defined using the `lambda` keyword

The function allows us to create small, one-line functions, often used for short tasks

Example:

```
>>> _
```

Special Functions ▶ Lambda Function

A lambda function is an anonymous function defined using the `lambda` keyword

The function allows us to create small, one-line functions, often used for short tasks

Example:

```
>>> odds = filter(lambda x: x % 2 != 0, range(11))
```

Special Functions ▶ Lambda Function

A lambda function is an anonymous function defined using the `lambda` keyword

The function allows us to create small, one-line functions, often used for short tasks

Example:

```
>>> odds = filter(lambda x: x % 2 != 0, range(11))
>>> _
```

Special Functions ▶ Lambda Function

A lambda function is an anonymous function defined using the `lambda` keyword

The function allows us to create small, one-line functions, often used for short tasks

Example:

```
>>> odds = filter(lambda x: x % 2 != 0, range(11))
>>> list(odds)
```

Special Functions ▶ Lambda Function

A lambda function is an anonymous function defined using the `lambda` keyword

The function allows us to create small, one-line functions, often used for short tasks

Example:

```
>>> odds = filter(lambda x: x % 2 != 0, range(11))
>>> list(odds)
[1, 3, 5, 7, 9]
>>> _
```

Special Functions

Special Functions ▶ `map()` Function

Special Functions ▶ `map()` Function

`map(f, seq)` returns the results of applying the function `f` to the items of `seq`

Special Functions ▶ `map()` Function

`map(f, seq)` returns the results of applying the function `f` to the items of `seq`

Example:

```
```

Special Functions ▶ `map()` Function

`map(f, seq)` returns the results of applying the function `f` to the items of `seq`

Example:

```
>>> _
```

Special Functions ▶ `map()` Function

`map(f, seq)` returns the results of applying the function `f` to the items of `seq`

Example:

```
>>> squares = map(lambda x: x ** 2, range(11))
```

Special Functions ▶ `map()` Function

`map(f, seq)` returns the results of applying the function `f` to the items of `seq`

Example:

```
>>> squares = map(lambda x: x ** 2, range(11))
>>> _
```

Special Functions ▶ `map()` Function

`map(f, seq)` returns the results of applying the function `f` to the items of `seq`

Example:

```
>>> squares = map(lambda x: x ** 2, range(11))
>>> list(squares)
```

Special Functions ▶ `map()` Function

`map(f, seq)` returns the results of applying the function `f` to the items of `seq`

Example:

```
>>> squares = map(lambda x: x ** 2, range(11))
>>> list(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> _
```