

Anonymous Functions: lambda

- Besides the def statement, Python also provides an expression form that generates function. Because of its similarity to a tool in the Lisp language, it's called lambda.
- Like def, this expression creates a function to be called later, but it returns the function instead of assigning it to a name. This is why lambdas are sometimes known as *anonymous (i.e., unnamed) functions*. *In practice, they are often used as a way to inline a function definition, or to defer execution of a piece of code.*

Anonymous Functions: lambda

```
def func(x, y, z): return x + y + z  
print(func(2, 3, 4))
```

```
f = lambda x, y, z: x + y + z  
print(f(2, 3, 4))
```

```
#output  
9  
9
```

Anonymous Functions: lambda

```
x = (lambda a="fee", b="fie", c="foe": a + b + c)
print(x("wee"))
```

```
#output
weefiefoe
```

Anonymous Functions: lambda

- lambda expressions introduce a local scope much like a nested def, which automatically sees names in enclosing functions, the module, and the built-in scope.

```
def knights():
    title = 'Sir'
    action = (lambda x: title + ' ' + x)
    return action                                # Return a function object

act = knights()
msg = act('robin')                             # 'robin' passed to x
print(msg)

#output

Sir robin
```

Anonymous Functions: lambda

- Generally speaking, lambda comes in handy as a sort of function shorthand that allows you to embed a function's definition within the code that uses it.

```
L = [lambda x: x ** 2,      # Inline function definition
      lambda x: x ** 3,
      lambda x: x ** 4]    # A list of three callable functions

for f in L:
    print(f(2))            # Prints 4, 8, 16

print(L[0](3))            # Prints 9

#output
4
8
16
9
```

Anonymous Functions: lambda

- In fact, you can do the same sort of thing with dictionaries and other data structures in Python to build up more general sorts of action tables.

```
key = 'got'
x={'already': (lambda: 2 + 2),
  'got':      (lambda: 2 * 4),
  'one':      (lambda: 2 ** 6)}[key]()
print(x)
```

```
#output
8
```

Anonymous Functions: lambda

- The fact that the body of a lambda has to be a single expression (not a series of statements) would seem to place severe limits on how much logic you can pack into a lambda. If you know what you're doing, though, you can code most statements in Python as expression-based equivalents.

```
lower = (lambda x, y: x if x < y else y)
print(lower('bb', 'aa'))
print(lower('aa', 'bb'))
```

```
#output
aa
aa
```

Anonymous Functions: lambda

- lambdas are the main beneficiaries of nested function scope lookup As a review, in the following the lambda appears inside a def—the typical case—and so can access the value that the name `x` had in the enclosing function's scope at the time that the enclosing function was called:

Anonymous Functions: lambda

```
def action(x):  
    return (lambda y: x + y) # Make and return function, 1  
  
act = action(99)  
print(act)  
print(act(2))                # Call what action returned  
  
action = (lambda x: (lambda y: x + y))  
act = action(99)  
print(act(3))  
print(((lambda x: (lambda y: x + y))(99))(4))  
  
#output  
    <function action.<locals>.<lambda> at 0x01367930>  
    101  
    102  
    103
```

Anonymous Functions: lambda

- One of the more common things programs do with lists and other sequences is apply an operation to each item and collect the results, selecting columns in database tables, incrementing pay fields of employees in a company, parsing email attachments, and so on.
- Python has multiple tools that make such collection-wide operations easy to code.
- For instance, updating all the counters in a list can be done easily with a for loop.

Anonymous Functions: lambda

```
counters = [1, 2, 3, 4]
updated = []
for x in counters:
    updated.append(x + 10)
print(updated)

def inc(x): return x + 10
print(list(map(inc, counters)))

print(list(map((lambda x: x + 10), counters)))

#output
[11, 12, 13, 14]
[11, 12, 13, 14]
[11, 12, 13, 14]
```

Filter and reduce

- The map function is a primary and relatively straightforward representative of Python's functional programming toolset. Its close relatives, filter and reduce, select an iterable's items based on a test function and apply functions to item pairs, respectively.

```
print(list(range(-5, 5)))  
print(list(filter((lambda x: x > 0), range(-5, 5))))
```

#output

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]  
[1, 2, 3, 4]
```

Filter and reduce

- The functional reduce call, is more complex. It accepts an iterable to process, but it's not an iterable itself—it returns a single result. Here are two reduce calls that compute the sum and product of the items in a list.

```
from functools import reduce
```

```
print(reduce((lambda x, y: x + y), [1, 2, 3, 4]))
```

```
print(reduce((lambda x, y: x * y), [1, 2, 3, 4]))
```

```
#output
```

```
10
```

```
24
```

x	y
1	2
5	3
6	4

Filter and reduce

```
def myreduce(function, sequence):  
    tally = sequence[0]  
    for next in sequence[1:]:  
        tally = function(tally, next)  
    return tally
```

```
print(myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5]))  
print(myreduce((lambda x, y: x * y), [1, 2, 3, 4, 5]))
```

#output

15
120

Handwritten diagram illustrating the reduce operation:

next	tally	new	sequence
2	1		[1, 2, 3, 4, 5]
3	2		
4	6		
5	24		
	120		

Formatting Expression and Printing

- Python defines the % binary operator to work on strings (you may recall that this is also the remainder of division, or modulus, operator for numbers). When applied to strings, the % operator provides a simple way to format values as strings according to a format definition. In short, the % operator provides a compact way to code multiple string substitutions all at once, instead of building and concatenating parts individually.
- The general structure of conversion targets looks like this:

*i = 5
j = 6*

`%[(keyname)][flags][width][.precision] typecode`

"the id is jd end" % (i, j) the 5 is 6 end

Typecode in Formatting

Code	Meaning
s	String (or any object's <code>str(X)</code> string)
r	Same as s, but uses <code>repr</code> , not <code>str</code>
c	Character (int or str)
d	Decimal (base-10 integer)
i	Integer
u	Same as d (obsolete: no longer unsigned)
o	Octal integer (base 8)
x	Hex integer (base 16)
X	Same as x, but with uppercase letters
e	Floating point with exponent, lowercase
E	Same as e, but uses uppercase letters
f	Floating-point decimal
F	Same as f, but uses uppercase letters
g	Floating-point e or f
G	Floating-point E or F
%	Literal % (coded as <code>%%</code>)

Formatting Expression and Printing

- The general structure of conversion targets looks like this:

`%[(keyname)][flags][width][.precision]typecode`

- Between the % and the type code character, you can do any of the following:
 - Provide a key name for indexing the dictionary used on the right side of the expression
 - List flags that specify things like left justification (-), numeric sign (+), a blank before positive numbers and a - for negatives (a space), and zero fills (0)
 - Give a total minimum field width for the substituted text
 - Set the number of digits (precision) to display after a decimal point for floating point numbers

Formatting Expression and Printing

```
*format1.py - C:/Documents and Settings/admin/Desktop/intro-python/examples/22-1/format1.py (3.4.4)*
File Edit Format Run Options Window Help

x = 1234
res = 'integers: ...%d...%-6d...%06d' % (x, x, x)
print(res)

x=1.23456789
print('%e | %f | %g' % (x, x, x))
print('%-6.2f | %05.2f | %+06.1f' % (x, x, x))

#output:
integers: ...1234...1234   ...001234
1.234568e+00 | 1.234568 | 1.23457
1.23      | 01.23 | +001.2
```

Formatting Expression and Printing

- When sizes are not known until runtime, you can use a computed width and precision by specifying them with a * in the format string to force their values to be taken from the next item in the inputs to the right of the % operator—the 4 in the tuple here gives precision:

```
print( '%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0))
```

0.333333, 0.33, 0.3333



%.4f

Dictionary-Based Formatting Expressions

```
print('%(qty)d more %(food)s' % {'qty': 1, 'food': 'spam'})
print()

reply = """
Greetings...
Hello %(name)s!
Your age is %(age)s
"""
values = {'name': 'Bob', 'age': 40}      # Build up values to substitute
print(reply % values)                  # Perform substitutions

#output:
1 more spam

Greetings...
Hello Bob!
Your age is 40
```

String Formatting Method Calls

- The string object's format method is based on normal function call syntax, instead of an expression. Specifically, it uses the subject string as a template, and takes any number of arguments that represent values to be substituted according to the template.
- Its use requires knowledge of functions and calls, but is mostly straightforward. Within the subject string, curly braces designate substitution targets and arguments to be inserted either by position (e.g., {1}), or keyword (e.g., {food}), or relative position ({}).

String Formatting Method Calls

```
template = '{0}, {1} and {2}'                                # By position
print(template.format('spam', 'ham', 'eggs'))
template = '{motto}, {pork} and {food}'                      # By keyword
print(template.format(motto='spam', pork='ham', food='eggs'))
template = '{motto}, {0} and {food}'                          # By both
print(template.format('ham', motto='spam', food='eggs'))
template = '{} , {} and {}'                                  # By relative position
print(template.format('spam', 'ham', 'eggs'))
```

#output:

```
spam, ham and eggs
spam, ham and eggs
spam, ham and eggs
spam, ham and eggs
```

String Formatting Method Calls

```
template = '{0}, {1} and {2}'                                # By position
print(template.format('spam', 'ham', 'eggs'))
template = '{motto}, {pork} and {food}'                     # By keyword
print(template.format(motto='spam', pork='ham', food='eggs'))
template = '{motto}, {0} and {food}'                         # By both
print(template.format('ham', motto='spam', food='eggs'))
template = '{} , {} and {}'                                 # By relative position
print(template.format('spam', 'ham', 'eggs'))
print()

print('{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2]))
X = '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
print(X)

#output:
spam, ham and eggs
spam, ham and eggs
spam, ham and eggs
spam, ham and eggs

3.14, 42 and [1, 2]
3.14, 42 and [1, 2]
```

Adding Keys, Attributes, and Offsets

- Like % formatting expressions, format calls can become more complex to support more advanced usage. For instance, format strings can name object attributes and dictionary keys—as in normal Python syntax, square brackets name dictionary keys and dots denote object attributes of an item referenced by position or keyword.


```

import sys

X='My {1[kind]} runs {0.platform}'.format(sys, {'kind': 'laptop'})
print(X)

X='My {map[kind]} runs {sys.platform}'.format(sys=sys, map={'kind': 'laptop'})
print(X)

somelist = list('SPAM')
print(somelist)

X='first={0[0]}, third={0[2]}'.format(somelist)
print(X)

X='first={0}, last={1}'.format(somelist[0], somelist[-1]) # [-1] fails in fmt
print(X)

parts = somelist[0], somelist[-1], somelist[1:3] # [1:3] fails in fmt
print('first={0}, last={1}, middle={2}'.format(*parts))

#output:

My laptop runs win32
My laptop runs win32

['S', 'P', 'A', 'M']

first=S, third=A
first=S, last=M

first=S, last=M, middle=['P', 'A']

```

```

print('{0:10} = {1:10}'.format('spam', 123.4567))
print('{0:>10} = {1:<10}'.format('spam', 123.4567))
print('{0.platform:>10} = {1[kind]:<10}'.format(sys, dict(kind='laptop'))))

print('{:10} = {:10}'.format('spam', 123.4567))
print('{:>10} = {:<10}'.format('spam', 123.4567))
print('{.platform:>10} = {[kind]:<10}'.format(sys, dict(kind='laptop'))))

print('{0:e}, {1:.3e}, {2:g}'.format(3.14159, 3.14159, 3.14159))
print('{0:f}, {1:.2f}, {2:06.2f}'.format(3.14159, 3.14159, 3.14159))

print('{0:X}, {1:o}, {2:b}'.format(255, 255, 255))
print(bin(255), int('11111111', 2), 0b11111111)
print(hex(255), int('FF', 16), 0xFF)
print(oct(255), int('377', 8), 0o377)

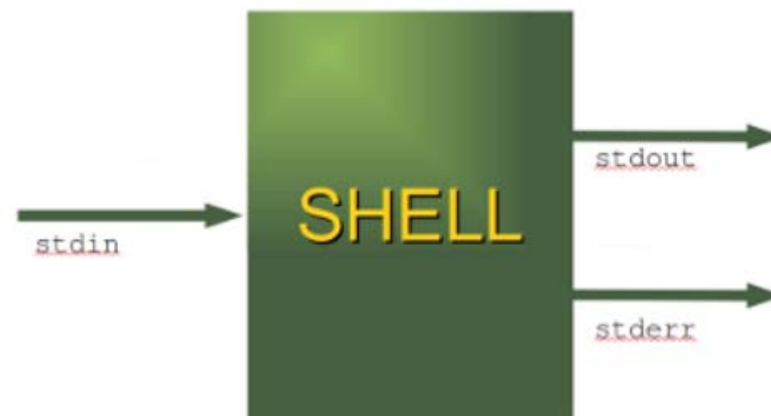
print('{0:.2f}'.format(1 / 3.0))
print('% .2f' % (1 / 3.0))
print('{0:.{1}f}'.format(1 / 3.0, 4))
print('%.*f' % (4, 1 / 3.0))

#output:
spam          = 123.4567
    spam      = 123.4567
    win32     = laptop
spam          = 123.4567
    spam      = 123.4567
    win32     = laptop
3.141590e+00, 3.142e+00, 3.14159
3.141590, 3.14, 003.14
FF, 377, 11111111
0b11111111 255 255
0xff 255 255
0o377 255 255
0.33
0.33
0.3333
0.3333

```

Standard input and output

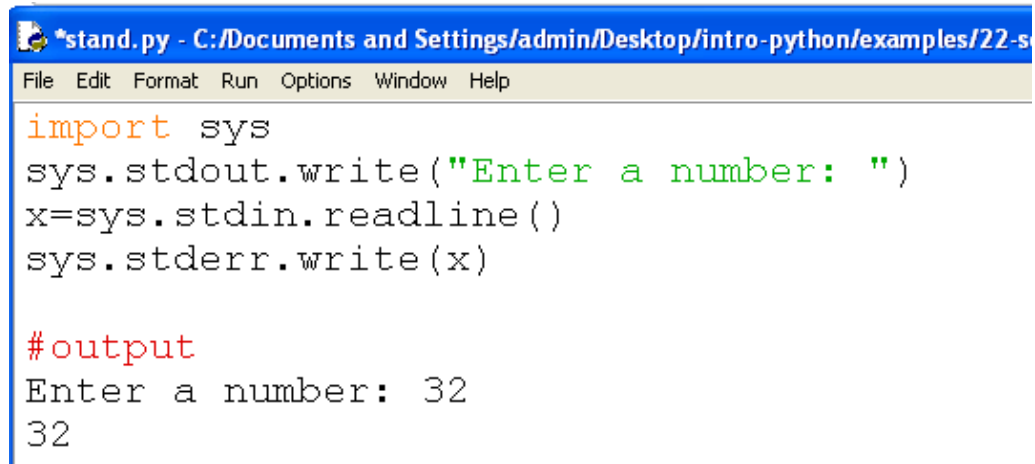
- Like many other languages, there are standard input, output, and error. These are in the `sys` module and are called `stdin`, `stdout`, and `stderr`. There are also immutable copies of these in `__stdin__`, `__stdout__`, and `__stderr__`.



Standard input and output

- The standard input (stdin) is normally connected to the keyboard, while the standard error and standard output go to the terminal (or window) in which you are working.
- There are some functions in stdin, stdout and stderr
 - stdin:
 - read
 - readline
 - readlines
 - stdout and err
 - write

Standard input and output



The screenshot shows a text editor window titled '*stand.py - C:/Documents and Settings/admin/Desktop/intro-python/examples/22-s'. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code in the editor is as follows:

```
import sys
sys.stdout.write("Enter a number: ")
x=sys.stdin.readline()
sys.stderr.write(x)

#output
Enter a number: 32
32
```

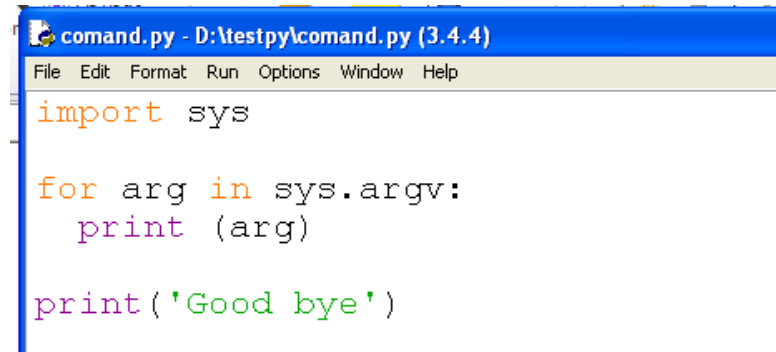
Command line arguments

- A *command-line argument* or parameter is an item of information provided to a program when it is started.

pprog 12 test

- Command-line arguments passed to a Python program are stored in `sys.argv` list. The first item in the list is name of the Python program, which may or may not contain the full path depending on the manner of invocation. `sys.argv` list is modifiable.

Command line arguments

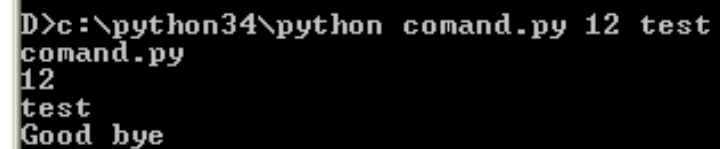


```
comand.py - D:\testpy\comand.py (3.4.4)
File Edit Format Run Options Window Help

import sys

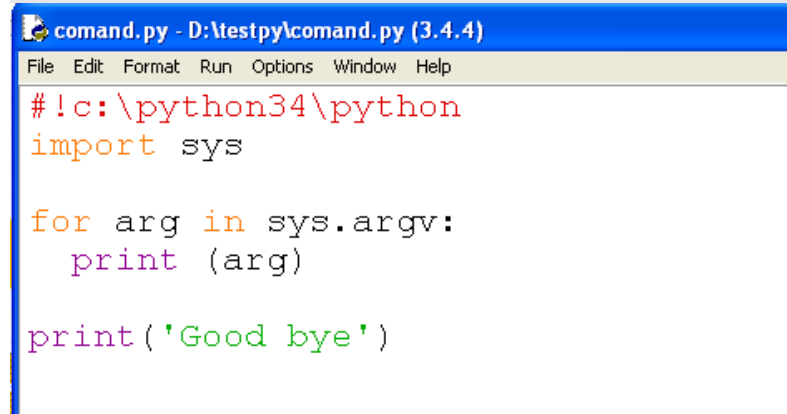
for arg in sys.argv:
    print (arg)

print('Good bye')
```



```
D>c:\python34\python comand.py 12 test
comand.py
12
test
Good bye
```

Command line arguments



```
comand.py - D:\testpy\comand.py (3.4.4)
File Edit Format Run Options Window Help
#!c:\python34\python
import sys

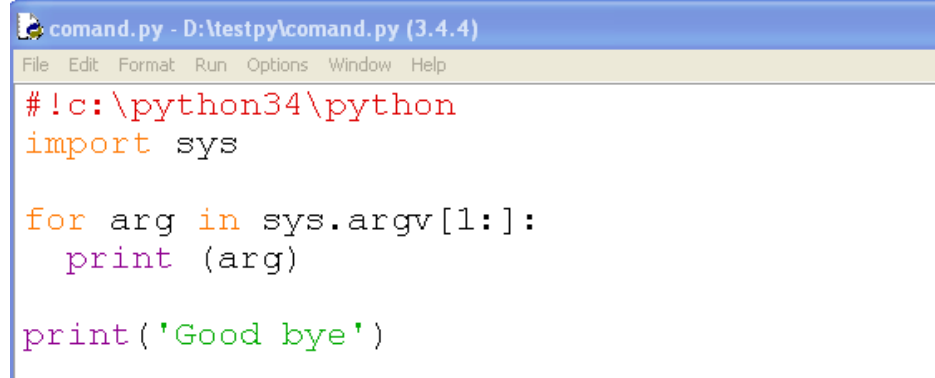
for arg in sys.argv:
    print (arg)

print('Good bye')
```



```
D:\testpy>comand.py 12 test 32
D:\testpy\comand.py
12
test
32
Good bye
```


Command line arguments



The screenshot shows a text editor window titled "comand.py - D:\testpy\comand.py (3.4.4)". The menu bar includes "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The code is as follows:

```
#!/c:\python34\python
import sys

for arg in sys.argv[1:]:
    print (arg)

print('Good bye')
```

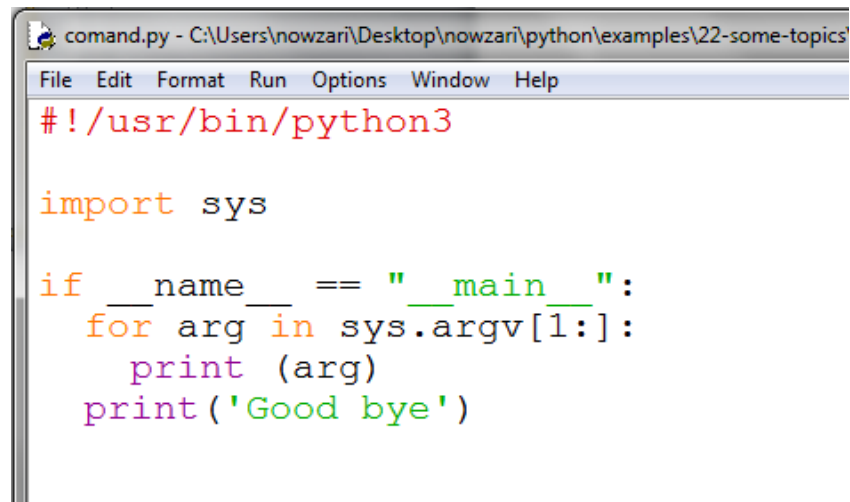


The screenshot shows a command prompt window with the following text:

```
D:\testpy>comand.py 12 test 32
12
test
32
Good bye
```

main function

- Each program has a main

A screenshot of a Python script editor window. The title bar shows the file path: "comand.py - C:\Users\nowzari\Desktop\nowzari\python\examples\22-some-topics". The menu bar includes "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The code is as follows:

```
#!/usr/bin/python3

import sys

if __name__ == "__main__":
    for arg in sys.argv[1:]:
        print (arg)
    print ('Good bye')
```

Palindrome Detector

- A palindrome is a string that reads the same when it is reversed.
- Punctuation, spaces, and capitalization are ignored.
- For example, the following is a palindrome:

A man, a plan, a canal, Panama!

```
#!/usr/bin/python3
import sys
def reverse(text):
    if len(text) <= 1:
        return text
    return reverse(text[1:]) + text[0]
def removePunc(text):
    C=''
    for char in text:
        if 'a'<= char <='z':
            C=C+char
    return(C)
def palindromtest(A):
    A=A.lower()
    A=removePunc(A)
    x=reverse(A)
    if x==A :
        return True
    else: return False

if __name__ == "__main__":
    A= sys.argv[1:][0]
    if palindromtest(A):
        print(A, 'is a palindrom')
    else:
        print(A, 'is not a palindrom')
    print('Good bye')
```

"this"
 "siht"
 "his"
 "sih"
 "is"
 "si"
 "s"

$A[::-1]$
 this siht
 A = amananam
 x amananam

Palindrome Detector

```
nowzari@nowzari:~/Desktop/python/examples/22-some-topics$ ./palin.py "test, test"
test, test is not a palindrom
Good bye
nowzari@nowzari:~/Desktop/python/examples/22-some-topics$ ./palin.py "A man, aplan, a canal, Panama!"
A man, aplan, a canal, Panama! is a palindrom
Good bye
nowzari@nowzari:~/Desktop/python/examples/22-some-topics$ ./palin.py "A man, aplan, a canal, Panam!"
A man, aplan, a canal, Panam! is not a palindrom
Good bye
```

Palindrome Detector

File Edit Format Run Options Window Help

```
#!/usr/bin/python3
import sys
def reverse(text):
    if len(text) <= 1:
        return text
    return reverse(text[1:]) + text[0]
def removePunc(text):
    C=''
    for char in text:
        if 'a'<= char <='z':
            C=C+char
    return(C)

A="test, sequense is not long!"
B=removePunc(A)
C=reverse(B)
print("input sequence is: ", A)
print("output sequence is: ", C)
print()
```

File Edit Format Run Options Window Help

```
#!/usr/bin/python3
import sys
from preverse1 import *

def palindromtest(A):
    A=A.lower()
    A=removePunc(A)
    x=reverse(A)
    if x==A :
        return True
    else: return False

A= sys.argv[1:][0]
if palindromtest(A):
    print(A, 'is a palindrom')
else:
    print(A, 'is not a palindrom')
print('Good bye')
```

Palindrome Detector

```
nowzari@nowzari:~/Desktop/p$ ./palinil.py "A man, aplan, a canal, panama!"  
input sequence is: test, sequence is not long!  
output sequence is: gnoltonsiesneugestset  
  
A man, aplan, a canal, panama! is a palindrom  
Good bye
```

Palindrome Detector

File Edit Format Run Options Window Help

```
#!/usr/bin/python3
import sys
def reverse(text):
    if len(text) <= 1:
        return text
    return reverse(text[1:]) + text[0]
def removePunc(text):
    C=''
    for char in text:
        if 'a'<= char <='z':
            C=C+char
    return(C)

if __name__ == "__main__":
    A="test, sequense is not long!"
    B=removePunc(A)
    C=reverse(B)
    print("input sequence is: ", A)
    print("output sequece is: ", C)
    print()
```

File Edit Format Run Options Window Help

```
#!/usr/bin/python3
import sys
from preverse import *

def palindromtest(A):
    A=A.lower()
    A=removePunc(A)
    x=reverse(A)
    if x==A :
        return True
    else: return False

if __name__ == "__main__":
    A= sys.argv[1:][0]
    if palindromtest(A):
        print(A, 'is a palindrom')
    else:
        print(A, 'is not a palindrom')
    print('Good bye')
```


Palindrome Detector

```
nowzari@nowzari:~/Desktop/p$ ./palini.py "A man, aplan, a canal, panama!"  
A man, aplan, a canal, panama! is a palindrom  
Good bye
```

End