Recursion

Recursion

There is another way to repeat executing sequences of instructions.

This is done by combining two of the three constructs to control the flow of execution.

Recursion

There is another way to repeat executing sequences of instructions.

This is done by combining two of the three constructs to control the flow of execution.

- Functions
- Conditionals
- Iteration

Recursion

There is another way to repeat executing sequences of instructions.

This is done by combining two of the three constructs to control the flow of execution.

- **Functions**
- **Conditionals**
- Iteration

Recursion

There is another way to repeat executing sequences of instructions.

For this, we will take a *self-referential* approach.

Self-reference?

Self-reference

The notion of *self-reference* is not just limited to computing. It actually arises in many fields (such as art, literature, mathematics, music, etc.).

Simple example.

This statement is false.

Russell's paradox

Consider categorizing all English adjectives into the following two kinds: autonymous and heteronymous. Can this be done?

- Autonymous means "self-describing". For example, "short" and "polysyllabic" are autonymous.
- Heteronymous means the opposite of autonymous. For example, "long" and "monosyllabic" are heteronymous.

Russell's paradox

Consider the word "heteronymous". This is neither autonymous nor heteronymous.

- If "heteronymous" were autonymous, then it would not be self-describing, meaning heteronymous.
- If "heteronymous" were heteronymous, then it would be self-describing, meaning autonymous.

Recursion in Python

In Python, recursion is realized by *self-referential* functions, i.e., *functions that call themselves*.

Write a function named `roo` that behaves like this:

```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
>>> roo(12345)
12345 1234 123 12 1
>>> roo(333)
333 33 3
```



```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

If a given integer has only 1 digit, consider:

```
>>> def roo1(n):
      print n

>>> roo1(8)
8
```

```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

If a given integer has only 2 digits, consider:

```
>>> def roo2(n):
      print n,
      print n/10

>>> roo2(87)
87 8
```



```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

If a given integer has only 2 digits, consider:


```
>>> def roo2(n):
      print n,
      roo1(n/10)

>>> roo2(87)
87 8
```



```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

If a given integer has only 3 digits, consider:


```
>>> def roo3(n):
      print n,
      print n/10
      print n/100

>>> roo3(876)
876 87 8
```



```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

If a given integer has only 3 digits, consider:


```
>>> def roo3(n):
      print n,
      roo2(n/10)

>>> roo3(876)
876 87 8
```

```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

If a given integer has only 4 digits, consider:

```
>>> def roo4(n):
      print n,
      roo3(n/10)

>>> roo4(8763)
8763 876 87 8
```



```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

In general, for *k*-digit integers, consider:

```
>>> def rook(n):
      print n,
      rook-1(n/10)
```



```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

Using self-reference, we can simplify as:

```
>>> def roo(n):
      print n,
      roo(n/10)
```



```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

Using self-reference, we can simplify as:

```
>>> def roo(n):
      print n,
      roo(n/10)
```

But, this does not work for 1-digit integers!

```
>>> roo(8763529)
8763529 876352 87635 8763 876 87 8
```

We treat 1-digit integers under special case:

```
>>> def roo(n):
      if n < 10:
        print n
      else:
        print n,
        roo(n/10)
```


Many recursive functions have the following basic structure.

```
def roo(<parameters>):
  if <boolean expression>:
    <base-case statements>
    <recursive statements>
```


Many recursive functions have the following basic structure.

```
def roo(<parameters>):
  if <boolean expression>:
    <base-case statements>
    <recursive statements>
```

Base-case statements must be *non-recursive*, i.e., it must not refer to the function itself.

Many recursive functions have the following basic structure.

```
def roo(<parameters>):
  if <boolean expression>:
    <base-case statements>
    <recursive statements>
```

Recursive statements refer to the function itself, but it must approach to the base case, typically involving "smaller" arguments.

For example, consider

```
>>> def roo(n):
      if n < 10:
        print n
      else:
        print n,
        roo(n/10)
```


For example, consider

```
>>> def roo(n):
      if n < 10:
        print n
      else:
        print n,
        roo(n/10)
```

Base-case statements must be *non-recursive*, i.e., it must not refer to the function itself.

For example, consider

```
>>> def roo(n):
      if n < 10:
        print n
      else:
        print n,
        roo(n/10)
```

The recursive statement approaches to the base case by considering a smaller integer, namely, `n/10`.