Computational Thinking and Algorithms

159.172

Recursive Functions

Amjed Tahir

a.tahir@massey.ac.nz

Previous contributors: Catherine McCartin

Creating functions

```
def fibs(num):
    assert (num > 1), "invalid parameter"
    result = [0,1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result
```

Creating functions

```
def fibs(num):
    assert (num > 1), "invalid parameter"
    result = [0,1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result
```

function = computational process which performs an action
and (usually) returns a value

Creating functions

```
def fibs(num):
    assert (num > 1), "invalid parameter"
    result = [0,1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result
```

function = computational process which performs an action
and (usually) returns a value

Creating functions

```
def fibs(num):
    assert (num > 1), "invalid parameter"
    result = [0,1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result
```

functions receive values to work with in the form of parameters

Creating functions

```
def fibs(num)
    assert (num > 1), 'invalid parameter'
    result = [0,1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result
```

you can use assert statements to deal with unacceptable parameters

Creating functions

```
def fibs(num)
    assert (num > 1), 'invalid parameter'
    result = [0,1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result
```

formal parameters = parameters in brackets after the function name in your def statement

Creating functions

```
def fibs(num)
    assert (num > 1), 'invalid parameter'
    result = [0,1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result

>>> fibs(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

actual parameters or arguments = values that you supply when you call an instance of your function

Creating functions

```
def fibs(num)
    assert (num > 1), 'invalid parameter'
    result = [0,1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result

>>> fibs(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

The binding of each formal parameter to an argument is kept in what is called a local scope.

Each time you call a function, a new scope or namespace is created.

Creating functions

```
def fibs(num):
    assert (num > 1), 'invalid parameter'
    result = [0,1]
    num = num - 2
    for i in range(num):
        result.append(result[-2] + result[-1])
    return result

>>> fibs(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

assigning a new value to a parameter inside a function

won't change the outside world

10 is still **10** !!!

Creating functions

```
def fibs(num):
    assert (num > 1), 'invalid parameter'
    result = [0,1]
    num = num - 2
    for i in range(num):
        result.append(result[-2] + result[-1])
    return result

>>> x = 10
>>> fibs(x)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

x is still **10** !!!

Creating functions

modifying a parameter that is bound to a mutable value inside a function can change the outside world

```
def change(n):
    n[0] = '159172'
    n = ['new', 'list']

>>> courses = ['old', 'list']
>>> change(courses)
>>> print(courses) => ???
```

Creating functions

modifying a parameter that is bound to a mutable value inside a function can change the outside world

```
def change(n):
    n[0] = '159172'
    n = ['new', 'list']

>>> courses = ['old', 'list']
>>> change(courses)
>>> courses['159172', 'list']
```

Creating functions

Inside the local scope of the function block, the first item in the list **n** is modified.
then the the parameter **n** is assigned to a new list value.
The "outside copy" of the **courses** argument still refers to the first list,
which was modified when we called **change(courses)**.

```
def change(n):
    n[0] = '159172'
    n = ['new', 'list']

>>> courses = ['old', 'list']
>>> change(courses)
>>> print(courses) => ???
```

Creating functions

Inside the local scope of the function block, the first item in the list **n** is modified.
then the the parameter **n** is assigned to a new list value.
The "outside copy" of the **courses** argument still refers to the first list,
which was modified when we called **change(courses)**.

```
def change(n):
    n[0] = '159172'
    n = ['new', 'list']

>>> courses = ['old', 'list']
>>> change(courses)
>>> courses['159172', 'list']
```

Creating functions

Now the parameter is assigned to a new list value and then that **new list** is modified, this doesn't change the "outside copy" of the **courses** argument, which still refers to the (unmodified) first list.

```
def change(n):
    n = ['new', 'list']
    n[0] = '159172'

>>> courses = ['old', 'list']
>>> change(courses)
>>> print(courses) => ???
```

Creating functions

Now the parameter is assigned to a new list value and then that **new list** is modified, this doesn't change the "outside copy" of the **courses** argument, which still refers to the (unmodified) first list.

```
def change(n):
    n = ['new', 'list']
    n[0] = '159172'

>>> courses = ['old', 'list']
>>> change(courses)
>>> courses['old', 'list']
```

Recursion

Recursion is a process where the solution to a problem depends on solutions to smaller instances of the same problem.

We first solve "simpler" versions of the problem that we have and then "do something" with the solutions to the simpler versions to get what we want.

Recursion

Recursion

A recursive solution to a problem must satisfy three rules:

1. A recursive solution must have a **base case**. The smallest version of the problem that cannot be further subdivided. **A solution is computed directly.**
2. A recursive solution must have a **recursive case**. This is a recursive call is made to a simpler or smaller version of the problem and then something is done with the result to the simpler version to get what we want.
3. A recursive solution must **make progress** towards the base case.

Recursive functions

A function can call any other function

A function can also call *an instance* **of itself.**

Each function call has its own **local scope**

- its own set of parameters
- its own local variables
- its own set of return values (if they're used)

Recursive functions

What happens when a function is called?

When `f()` calls `g()`,
the system:

- **saves local scope of f** (all the local variables and parameters)
- uses variables from **f** as parameters in the call to **g**
- jumps to first instruction of **g**, and executes that function
- returns from **g**, passing any returned value(s) to back to **f**
- **restores the saved local scope of f**
- resumes execution in **f** just after the function call to **g**

Recursive functions

A function can call any other function.

A function can also call an instance of itself.

When a function "calls itself " a new instance of the function **with its own *scope* (or *namespace*) is created**.

The result of this call will be "passed back" to the calling function when the function exits, exactly the same as the result of any other function.

A *recursive* function call is a separate entity (has its own locals and parameters) just like any other function call.

Recursive functions

The important thing to think about is what parameters we need to allow the recursion to work.

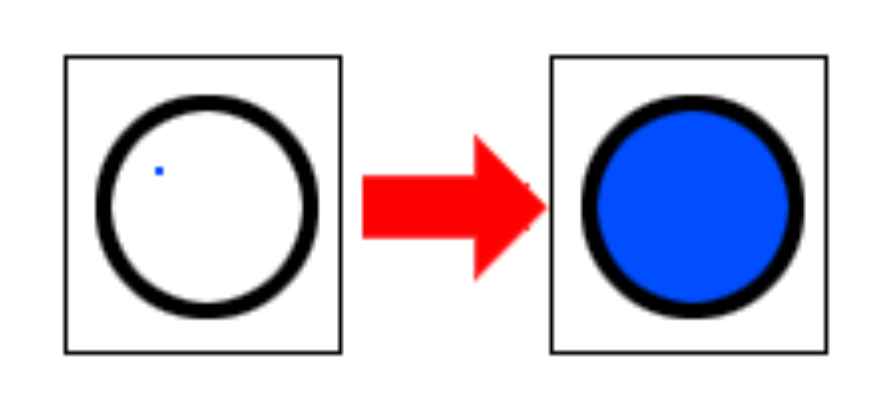
```
def countdown(n):
        print(n)
        if n == 0:  # this is the base case
           return   # return,this is the "Base caes"
         countdown(n - 1)  # the recursive call

>>> countdown(3)
3
2
1
0
```

floodfill function

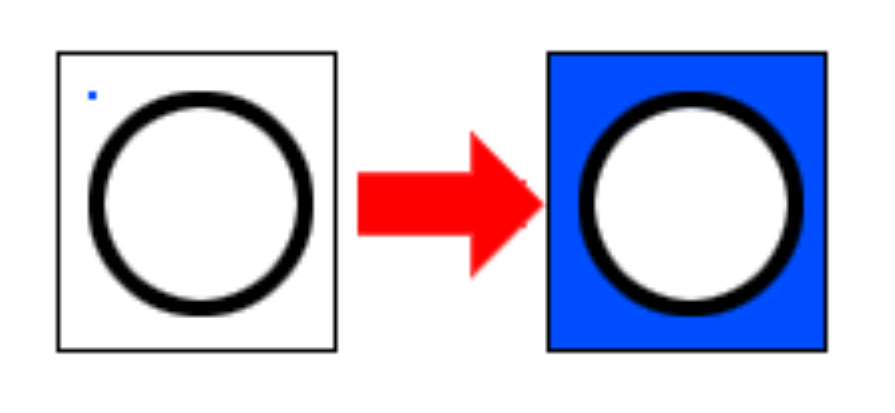
Flood filling can be used when you want to change the color of an area in an image.

The blue flood filling starts at the blue point (which was originally white), so it will keep spreading as long as it finds adjacent white pixels. Imagine it as blue paint pouring on that dot, and it keeps spreading until it hits any non-white colors (like the black circle).



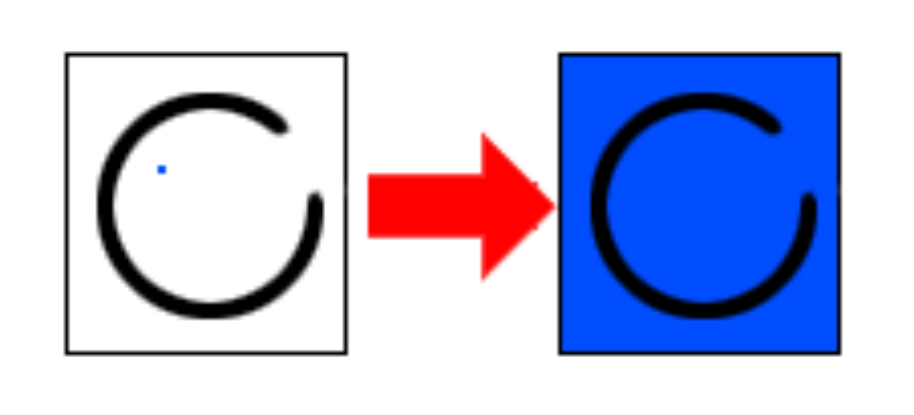
floodfill function

If we started the flood filling from the outside of the circle, then the entire outside area would be filled up instead:



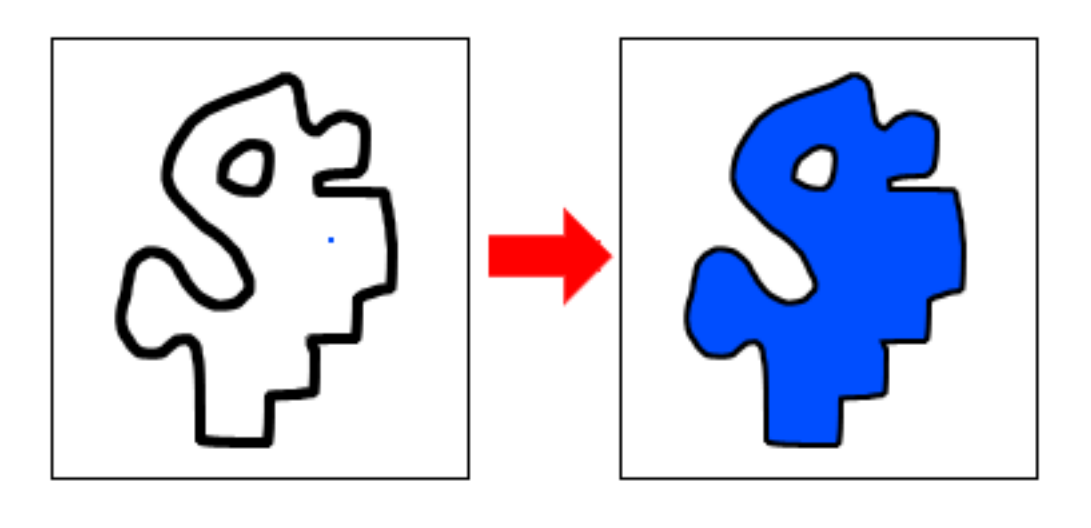
floodfill function

Flood filling a circle that is not completely enclosed wouldn't work the same way. The flood filling color would "leak out" the open space in the circle and then fill up the outside space as well:



floodfill function

The clever thing about flood fill is that it can fill up any arbitrary enclosed shape:



floodfill function

What parameters do we need to allow the recursion to work?

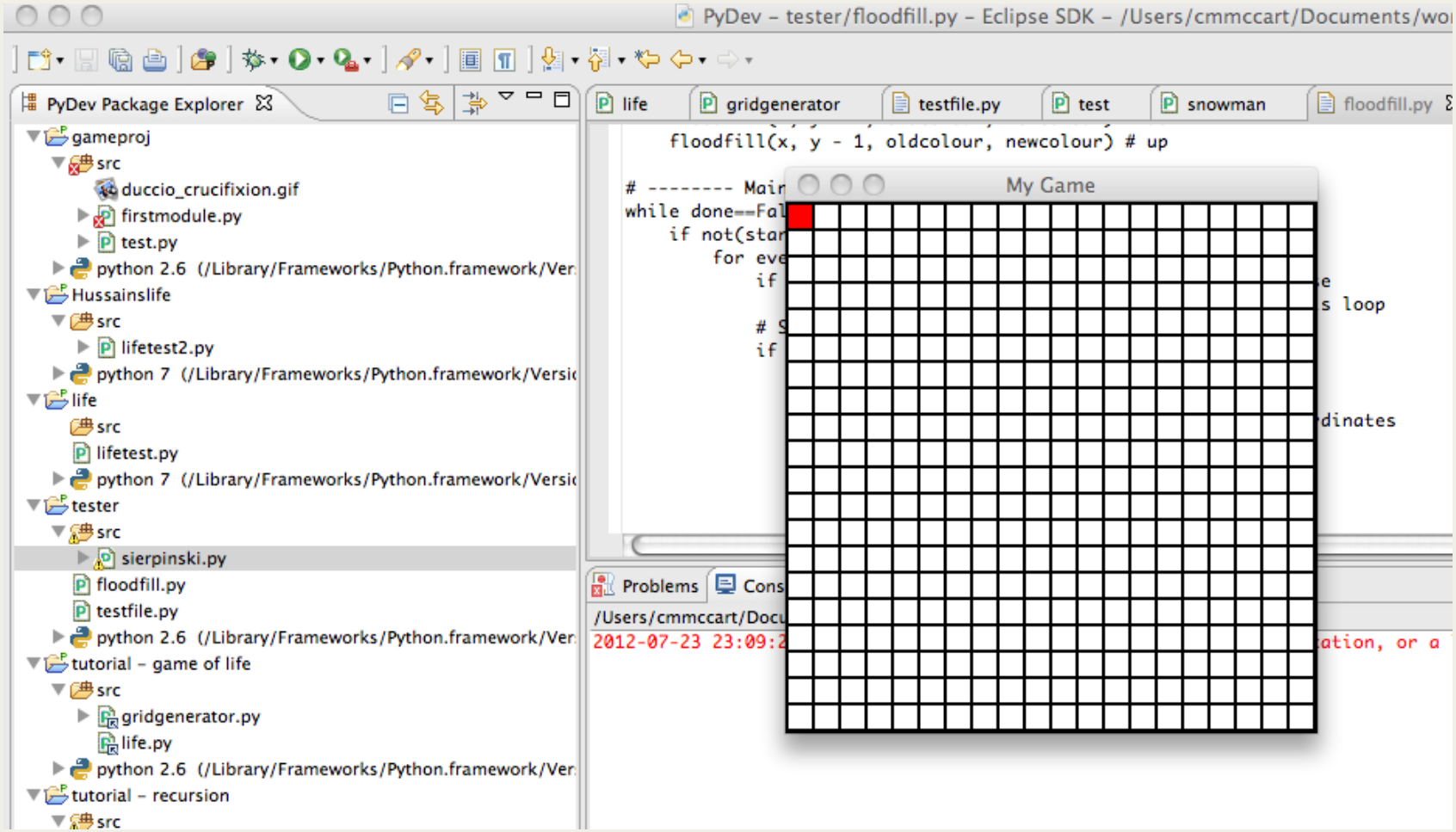
```
def floodfill(x, y, oldcolour, newcolour):
    # don't recurse out of the grid!
    if (x < 0) or (x > 19) or (y < 0) or (y > 19):
        return
    # the base case
    if grid[x][y] != oldcolour:
        return
    # the recursive case
    grid[x][y] = newcolour
    floodfill(x + 1, y, oldcolour, newcolour) # right
    floodfill(x - 1, y, oldcolour, newcolour) # left
    floodfill(x, y + 1, oldcolour, newcolour) # down
    floodfill(x, y – 1, oldcolour, newcolour) # up
```

floodfill function

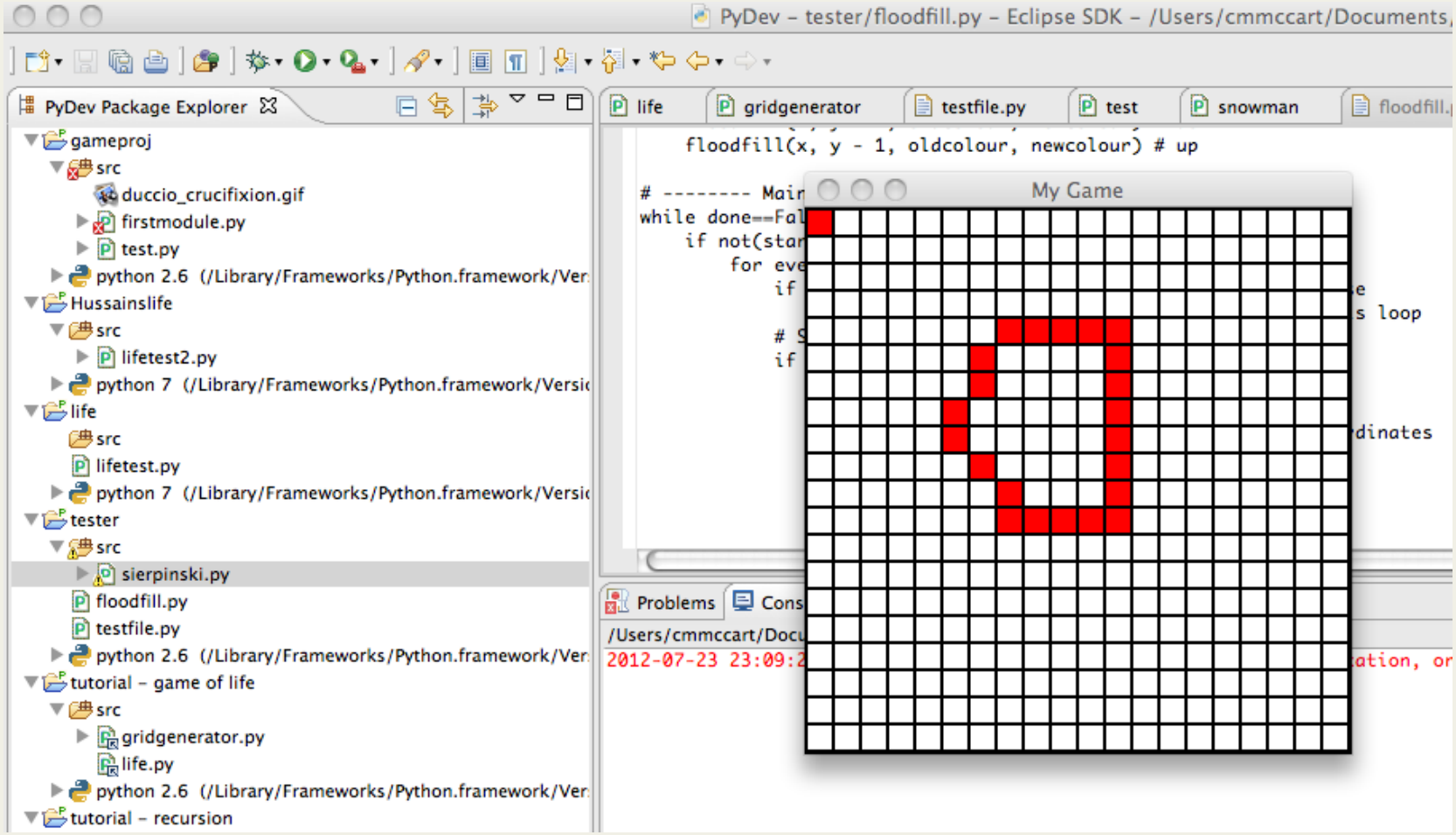
```
def floodfill(x, y, oldcolour, newcolour):
    # don't recurse out of the grid!
    if (x < 0) or (x > 19) or (y < 0) or (y > 19):
        return
    # the base case
    if grid[x][y] != oldcolour:
        return
    # the recursive case
    grid[x][y] = newcolour
    floodfill(x + 1, y, oldcolour, newcolour) # right
    floodfill(x - 1, y, oldcolour, newcolour) # left
    floodfill(x, y + 1, oldcolour, newcolour) # down
    floodfill(x, y – 1, oldcolour, newcolour) # up
```

Function call: `floodfill(10, 10, 0, 1)`

floodfill function



floodfill function



floodfill function