	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
9:00 - 10:00					
10:00 - 11:00			Consultation ICT3.20	INFT1004 Lab 4 *ICT3.44* Will	
11:00 - 12:00			INFT1004 Lab 1 - BYOD *ICT3.29* Keith		
12:00 - 1:00				INFT1004 Lab 5 *ICT3.44* Will	
1:00 - 2:00			PASS MCG 29		
2:00 - 3:00		PASS W 238	INFT1004 Lab 2 *ICT3.37* Brendan	INFT1004 Lab 5 *ICT3.44* Will	
3:00 - 4:00		INFT1004 Lecture GP 201			
4:00 - 5:00			INFT1004 Lab3 *ICT3.44* Brendan	INFT1004 Lab 6 *ICT3.44* Will	
5:00 - 6:00					
6:00 - 7:00					
7:00 - 8:00					

INFT1004 - SEMESTER 1 - 2017		LECTURE TOPICS	
Week 1	Feb 27	Introduction, Assignment, Arithmetic	
Week 2	Mar 6	Sequence, Quick Start, Programming Style	
Week 3	Mar 13	Pictures, Functions, Media Paths	
Week 4	Mar 20	Arrays, Pixels, For Loop, Reference Passing	
Week 5	Mar 27	Nested Loops, Selection, Advanced Pictures	Practical Test
Week 6	Apr 3	Lists, Strings, Input & Output, Files	
Week 7	Apr 10	Drawing Pictures, Program Design, While Loop	Assignment set
Recess	**Apr 14 – Apr 23**	**Mid Semester Recess Break**	
Week 8	Apr 24	No Lecture / Revision and Assignment in Labs	
Week 9	May 1	Data Structures, Processing sound	
Week 10	May 8	Advanced sound	Assignment part 1 due 8:00am Tue, May 9
Week 11	May 15	Movies, Scope, Import	
Week 12	May 22	Turtles, Writing Classes	Assignment part 2 due 8:00am Tue, May 23
Week 13	May 29	Revision	
Mid Year Examination Period - MUST be available normal & supplementary period			

Lecture Topics and Lab topics are the same for each week

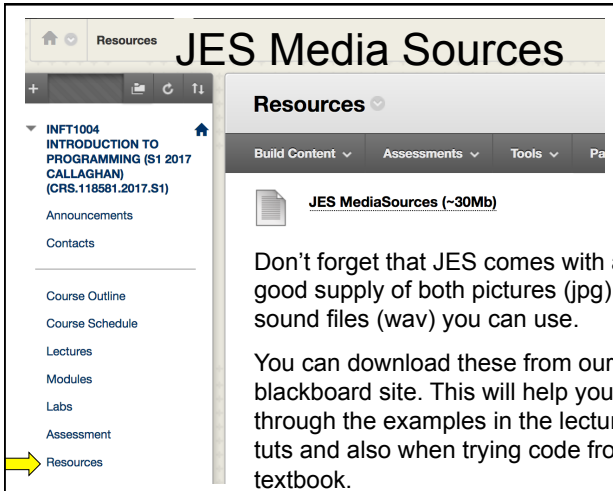
INFT1004

Introduction to Programming

Module 3.1
Pictures, Pixels, Colors

Guzdial & Ericson - Third Edition – chapter 3
Guzdial & Ericson - Fourth (Global) Edition – chapter 4

JES Media Sources



Don't forget that JES comes with a good supply of both pictures (jpg) and sound files (wav) you can use.

You can download these from our blackboard site. This will help you work through the examples in the lectures, tuts and also when trying code from the textbook.

Text Book

You should now have a copy of the text book. In the following modules we will often reference back to the code and examples in the text book.

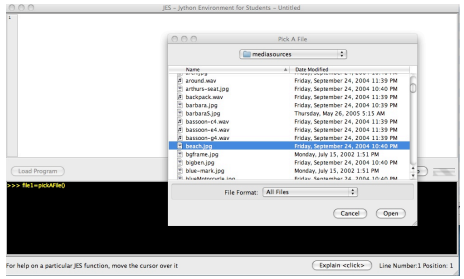
You should (if you haven't already) be working through chapters 1, 2, 3, 4 of the textbook.

JES and Pictures

try

```
>>> pictureFile = pickAFile()
```

Choose a picture file, eg beach.jpg, from the mediasources folder that goes with the textbook

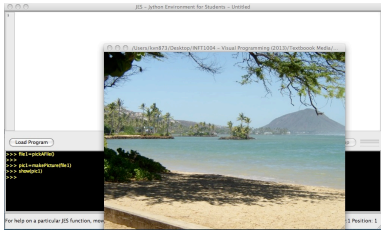


You now have a variable called `pictureFile` of type string

JES and Objects

```
>>> myPicture = makePicture(pictureFile )
```

You now have an object called `myPicture`, and that object is a picture



Show and Explore

Note you will have to type

```
>>> show(myPicture)
```

to see the picture

```
show(..)
```

is another function that comes with JES and it is very useful for showing pictures!

Show and Explore

Note you will have to type

```
>>> show(myPicture)
```

to see the picture

Remember: We can do this in the command window by typing a lot of commands (functions) **or** we can write a similar function in the program section.

JES and Objects

```
def testShowPicture():

    #select a jpg file from JES Media Source
    pictureFile = pickAFile()

    #You now have to turn the file path into a Picture
    myPicture = makePicture(pictureFile)

    #"Picture" is a special type (class) in JES - print it
    print(myPicture)

    #now show it
    show(myPicture)
```

Mod3_1_ColorsPictures.py

getWidth(), getHeight()

```
def testShowPictureDetails ():

    pictureFile = pickAFile()

    #You now have to turn the file path into a Picture
    myPicture = makePicture(pictureFile)
    print(myPicture)

    #Find out the size of the picture (number of pixels)
    width = getWidth(myPicture)
    height = getHeight(myPicture)
    print("height=" + str(height) + "   width=" + str(width))
```

Mod3_1_ColorsPictures.py

Show and Explore

try

```
>>> explore(myPicture)
```

and see what this does

`explore(..)`

is another function that comes with JES and it is very useful for exploring pictures!

Show and Explore

try

```
>>> explore
```

and see what

Remember: We can do this in the command window by typing a lot of commands (functions) **or** we can write a similar function in the program section.

JES and Objects

```
def testExplorePicture():

    #select a jpg file from JES Media Source
    pictureFile = pickAFile()

    #You now have to turn the file path into a Picture
    myPicture = makePicture(pictureFile)

    #"Picture" is a special type (class) in JES - print it
    print(myPicture)

    #explore it
    explore(myPicture)
```

Mod3_1_ColorsPictures.py

Remember Types

We've met simple types like integers, strings, floats, (boolean later)

`Picture` is one example of a more complex type (class)

Different types (classes) can have different things done with them

`Picture`

Remember Types

We've met simple types like integers, strings, floats, booleans

plex

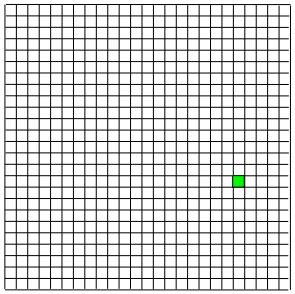
t

For example, you can show and explore pictures. See how the explore tool lets you check out each individual pixel of a picture

```
>>> explore(myPicture)
>>> show(myPicture)
>>> height = getHeight(myPicture)
>>> width = getWidth(myPicture)
```

More about Pictures

On a computer pictures are made up of a collection of small elements called pixels.

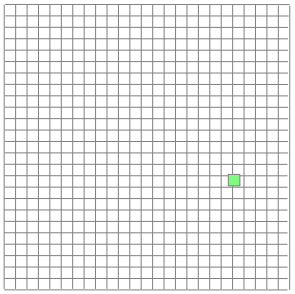


More about Pictures

On a computer pictures are made up of a collection of small elements called pixels.

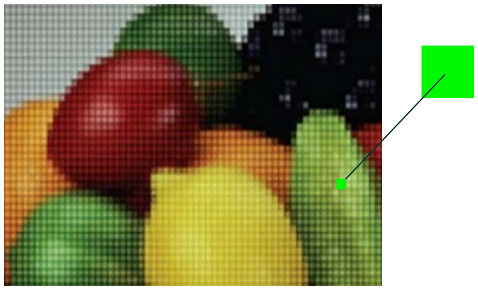
`Pixel`

is another special type (class) in JES



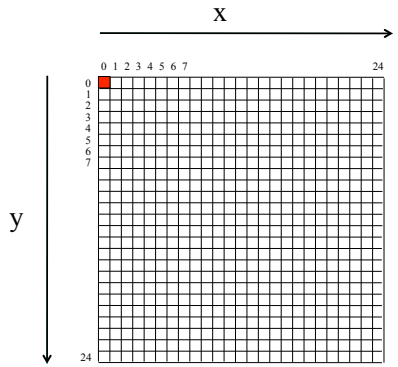
Pictures and Pixels

Pixels have a position (x,y) in the picture and they store the color at that position.



x= 59
y = 27
red = 0
green = 255
blue = 0

Pixels



Pixels in a picture start in top left corner at x=0 and y=0

getPixel

```
aPixel = getPixel(myPicture, 7, 4)
```

a Picture

x (integer)

y (integer)

`myPic, 7, 4`
are the arguments needed when we call the function

We need these because someone who wrote the function, `getPixel` decided they these parameters were important.

getPixel

```
aPixel = getPixel(myPicture, 7, 4)
```

Pixel

Picture

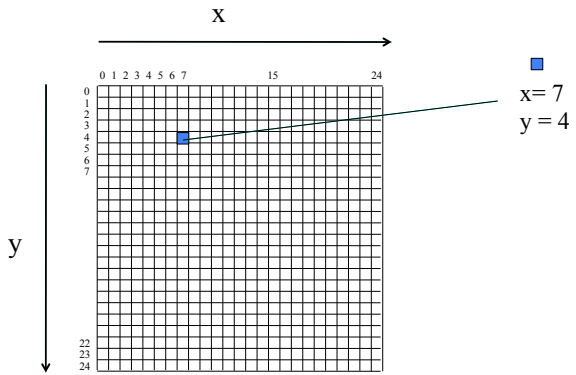
x (integer)

y (integer)

The function, `getPixel` returns a `Pixel`

getPixel() function

- `aPixel = getPixel(myPicture, 7, 4)`



Names and types

```
aPixel = getPixel(myPicture, 7, 4)
```

Every name refers to an item of a particular type
`aPixel` is an item (object) of type `Pixel`

An easy way to find out is just to try it!

```
>>> print aPixel
```

Color : Another class in JES

`Color`

`Color` is another class in JES

```
aPixel = getPixel(myPicture, 7, 4)

pixelColor = getColor(aPixel)

print pixelColor
```


Playing with Pixels

We can name individual pixels in the picture and set their color

```
pixel1 = getPixel(myPicture, 32, 32)
pixel2 = getPixel(myPicture, 33, 33)
setColor(pixel1, yellow)
setColor(pixel2, blue)
explore(myPicture)
```

Mod3_1_ColorsPictures.py - playWithPixels(aPicture)

Playing with Pixels

We can name individual pixels in the picture and set their color

```
pixel1 = getPixel(myPicture, 32, 32)
pixel2 = getPixel(myPicture, 33, 33)
setColor(pixel1, yellow)
setColor(pixel2, blue)
explore(myPicture)
```

yellow and **blue** are `Colors` defined in JES

Playing with Pixels

We can name individual pixels in the picture and set their color

```
pixel1 = getPixel(myPicture, 32, 32)
pixel2 = getPixel(myPicture, 33, 33)
setColor(pixel1, yellow)
setColor(pixel2, blue)
```

Notice that when you explore the picture again, you get a new window; explore doesn't have a refresh option

Working with Colors

So another complex type in JES is `Color`
(with the US spelling)

we can try..

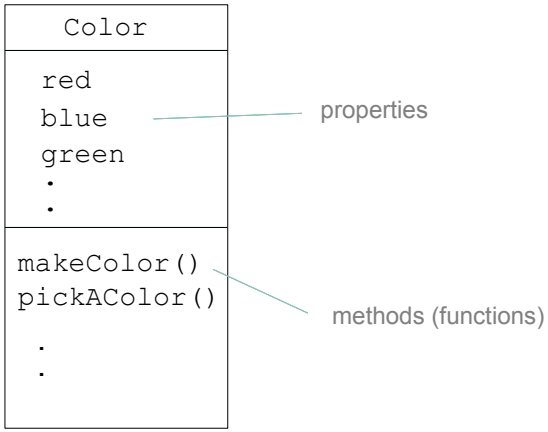
```
myColor = getColor(aPixel)

print myColor
```


Working with Colors

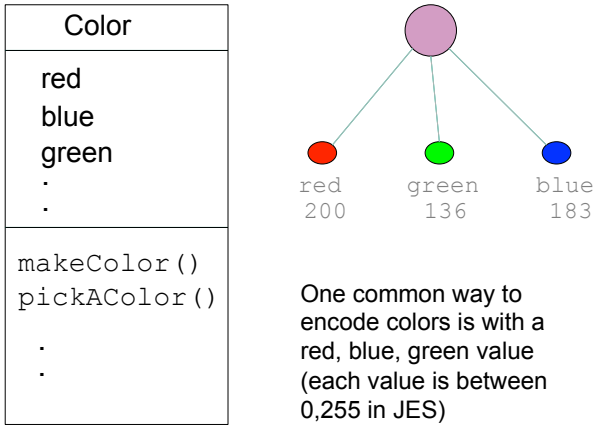
we can also try..

```
myRed = getRed(aPixel)

print myRed

#getGreen, getBlue
```


Working with Colors



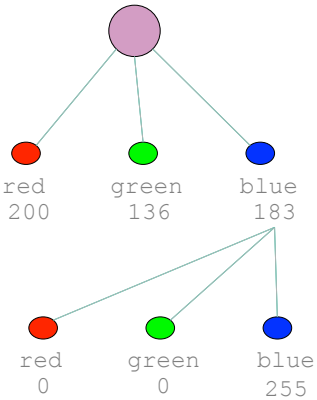
Working with Colors



One common way to encode colors is with a red, blue, green value (each value is between 0,255 in JES)

Working with Colors

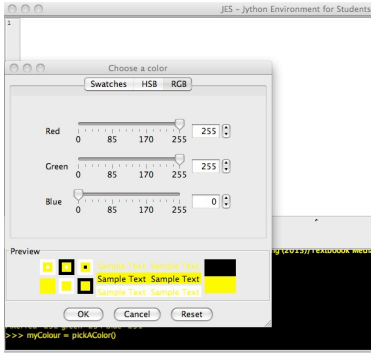
Color
red blue green . .
makeColor() pickAColor() . .



Working with Colors

```
>>> myColor = pickAColor()
```

We can do a lot of things with colours once we understand how colours can be expressed as red, green, and blue values with each value in the range 0 to 255



pickAColor()

```
myColor = pickAColor()

print(myColor)

myPixel= getPixel(myPicture, 10, 40)

setColor(myPixel, myColor)

explore(myPicture)
```

Mod3_1_ColorsPictures.py - playWithColors(aPicture)

makeColor (redInt, greenInt, blueInt)

```
myColor = makeColor(255,255,0)

print(myColor)

myPixel= getPixel(myPicture, 10, 40)

setColor(myPixel, myColor)

explore(myPicture)

myColor = pickAColor()
```

Mod3_1_ColorsPictures.py - playWithColors(aPicture)

Color channels

```
getRed()
getGreen()
getBlue()
```

find the individual channels of a pixel

Color channels

```
getRed()
getGreen()
getBlue()
```

find the individual channels of a pixel

```
setRed()
setGreen()
setBlue()
```

alter the individual channels of a pixel

Color channels

```
getRed()
getGreen()
getBlue()
```

find the individual channels of a pixel

```
setRed()
setGreen()
setBlue()
```

alter the individual channels of a pixel

Color channels

```
getRed()
getGreen()
getBlue()
```

gets the green value of the pixel (between 0..255)

```
pixelsGreenPart = getGreen(myPixel)
print pixelsGreenPart
```

Color channels

sets the red value of the pixel to 0 (value must be between 0..255)

```
setRed()
setGreen()
setBlue()
```

```
setRed(myPixel, 0)
print myPixel
```

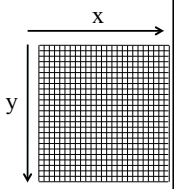

A program

```
def yellowLine(aPicture):
    ### This function draws a yellow line of pixels
    ### on the picture by setting 5 pixels to yellow

    aPixel = getPixel(aPicture, 40, 30)
    setColor(aPixel , yellow)
    aPixel = getPixel(aPicture, 41, 30)
    setColor(aPixel , yellow)
    aPixel = getPixel(aPicture, 42, 30)
    setColor(aPixel , yellow)
    aPixel = getPixel(aPicture, 43, 30)
    setColor(aPixel , yellow)
    aPixel = getPixel(aPicture, 44, 30)
    setColor(aPixel , yellow)

    explore(aPicture)
```


A program

```
def yellowLine(aPicture):
    ### This function draws a yellow line of pixels
    ### on the picture by setting 5 pixels to yellow

    aPixel = getPixel(aPicture, 40, 30)
    setColor(aPixel , yellow)
    aPixel = getPixel(aPicture, 41, 30)
    setColor(aPixel , yellow)
    aPixel = getPixel(aPicture, 42, 30)
    setColor(aPixel , yellow)
    aPixel = getPixel(aPicture, 43, 30)
    setColor(aPixel , yellow)
    aPixel = getPixel(aPicture, 44, 30)
    setColor(aPixel , yellow)
```

Notice they all have the same y value (30) and slightly different x values (40, 41,42,43,44) so it makes a horizontal row of yellow pixels



Remember – Save and Load

The program's written in the program area, but JES can't yet run it

First you have to Load the program; this tells JES to take note of any functions defined in the program and be ready to use them

And before you load the program you must save it!

Saving the program

Choose a sensible location

Give the program a sensible name

Give the name a .py extension

(On the Macs in the lab it's usually easiest to save to the desktop – as JES has trouble seeing USB drives etc – just a bug in JES – remember to copy any changes back to your USB and don't leave your USB behind!)

Trying the program

```
def yellowLine(aPicture):
```

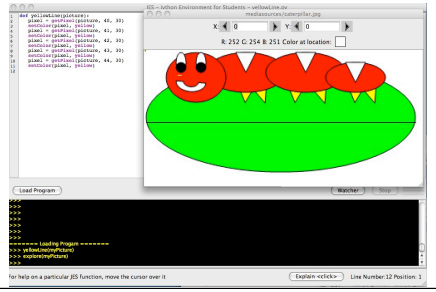
Picture

I defined the function to need a picture as a parameter – so I'll need to get a real picture to pass in as an argument – if I want to try it

Trying the program

set it to use the JES Media Resources

```
>>> setMediaPath()
>>> pictureFile = getMediaPath("caterpillar.jpg")
>>> myPicture = makePicture(pictureFile)
>>> yellowLine(myPicture)
```



Arguments and Parameters

```
def yellowLine(aPicture):
   aPixel = getPixel(aPicture, 40, 30)
   setColor(aPixel , yellow)
   .
   .
   .
```

So, parameters are used to specify the things that a function needs (these are decided by the programmer when they write the function)

Arguments and Parameters

```
def yellowLine(aPicture):
   aPixel = getPixel(aPicture, 40, 30)
   setColor(aPixel , yellow)
   .
   .
   .
```

For example, when I wrote the `yellowLine` function, I included a single parameter and I called it **`aPicture`**.

Arguments and Parameters

parameter

```
def yellowLine(aPicture):
   aPixel = getPixel(aPicture, 40, 30)
   setColor(aPixel , yellow)
   .
   .
   .
```

A parameter is a name that is used within the function definition to represent the argument (which only gets supplied later)

Arguments and Parameters

parameter

```
def yellowLine(aPicture):
   aPixel = getPixel(aPicture, 40, 30)
   setColor(aPixel , yellow)
   .
   .
   .
```

Then in the body of the function we use the name **`aPicture`** when we want to refer to the parameter

Arguments and Parameters

parameter

```
def yellowLine(aPicture):
.
   .
   .

>>> myFile = pickAFile()
>>> myPicture= makePicture(myFile)
>>> yellowLine(myPicture)
```

argument

The arguments are expressions provided when the function is called.

Arguments and Parameters

parameter

```
def yellowLine(aPicture):
.
    .
    .
```

```
>>> yellowLine(myPicture)
```

argument

When we use (call) a function with parameters, we need to include arguments that correspond to each parameter

Arguments and Parameters

How does that work?

The parameter is a placeholder

When writing the function, the programmer needs to say 'do this with the argument that's passed in'

But the programmer doesn't know what the actual argument will be called; it could be myFile, file1, fred, etc

Arguments and Parameters

How does that work?

So the programmer uses a parameter when writing the function.

When the function is called, the parameter (eg `aPicture`) becomes another name for the argument (eg `myPicture`)

Then whatever the programmer said should be done with the parameter (`aPicture`) is done with argument (`myPicture`).

Some Classes

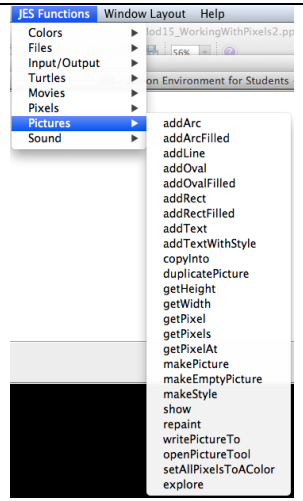
We have met some new classes which we will be using over the next few weeks. JES has lost of functions that work with each class

`Picture`

`Color`

`Pixel`

The text book – chapters 1,2,4 also introduce these concepts –read!

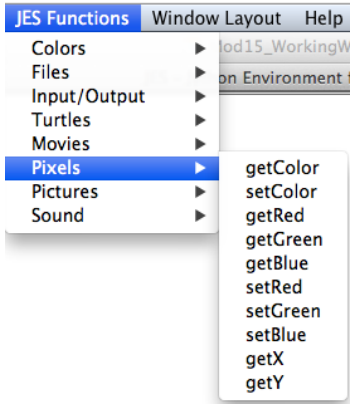


Picture

```
show(..)
explore(..)
makePicture(..)
getHeight(..)
getWidth(..)
getPixel(..)
.
.
.
```

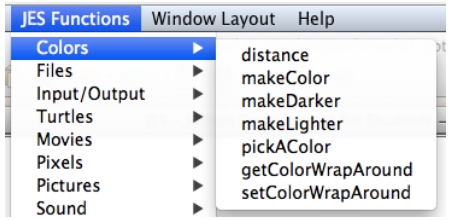

Pixel

```
getColor(..)
setColor(..)
getRed(..)
getGreen(..)
getBlue(..)
setRed(..)
setGreen(..)
setBlue(..)
.
.
```



Color

```
pickAColor(..)
makeColor(..)
.
.
.
```



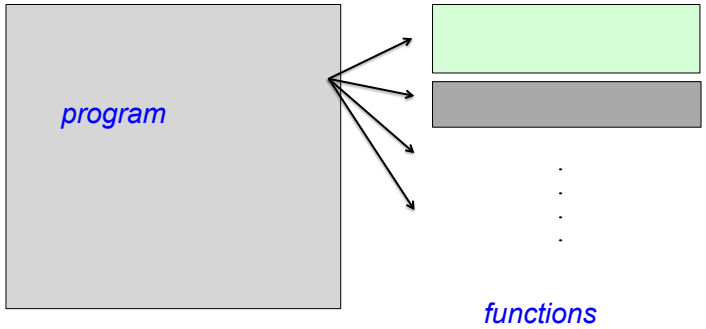
INFT1004

Introduction to Programming

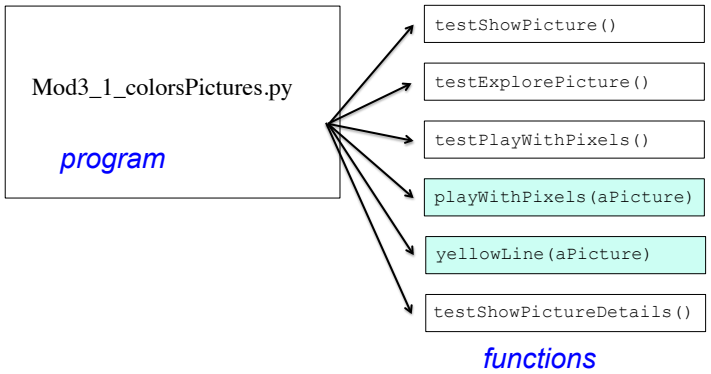
Module 3.2
Functions

Programs in JES

A program consists of one or more functions



Programs in JES



Mod3_1_colorsPictures.py

Programs in JES

This is generally the best way to work in JES (eg on Tutorial or assignment problems) – Create a program and write one, or more, functions to solve the problem(s)

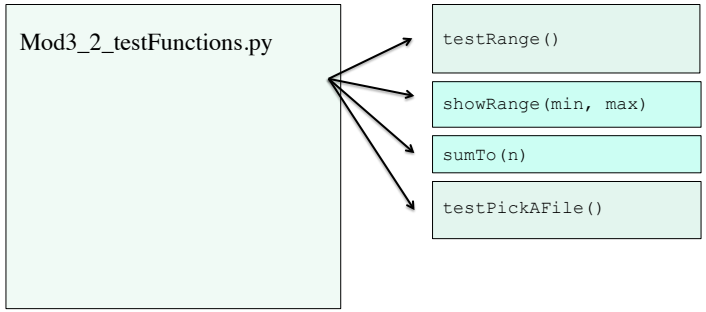
Mod3_1_colorsPictures.py

Programs in JES

You will also see (especially in lab solutions) that I often write one function that solves the problem and then a second function that tests the solution

Mod3_1_colorsPictures.py

More Programs in JES



Mod3_2_testFunctions.py

Programs in JES

```
def testRange():
  #test the range function
  showRange(-2,5)
  showRange(4,10)

  start = 3
  end = start + 5 + 1
  showRange(start, end)

def showRange(min, max):
  #This function plays with the range functions
  #it uses the print statement to print the results

  myRange = range(min, max+1)
  print myRange
```

This function calls the showRange function 3 times

Mod3_2_testFunctions.py

Programs in JES

```
def showRange(min, max):
  myRange = range(min, max+1)
  print myRange
```

Every function has the following structure

- the word 'def' (for define)
- a name – a meaningful and informative name
- a pair of parentheses, possibly empty, possibly with arguments
- a colon (indicating that more is to follow)
- a body – the instructions to say what the function should do – this must be indented from the def

Programs in JES

```
def showRange(min, max):
  myRange = range(min, max+1)
  print myRange
```

Every function has the following structure

- the word 'def' (for define)
- a name – a meaningful and informative name
- a pair of parentheses, possibly empty, possibly with arguments
- a colon (indicating that more is to follow)
- a body – the instructions to say what the function should do – this must be indented from the def

Programs in JES

```
def showRange(min, max):
  myRange = range(min, max+1)
  print myRange
```

Every function has the following structure

- the word 'def' (for define)
- a name – a meaningful and informative name
- a pair of parentheses, possibly empty, possibly with arguments
- a colon (indicating that more is to follow)
- a body – the instructions to say what the function should do – this must be indented from the def

Programs in JES

```
def showRange(min, max):
  myRange = range(min, max+1)
  print myRange
```

Every function has the following structure

- the word 'def' (for define)
- a name – a meaningful and informative name
- a pair of parentheses, possibly empty, possibly with arguments
- a colon (indicating that more is to follow)
- a body – the instructions to say what the function should do – this must be indented from the def

Programs in JES

```
def showRange(min, max):
  myRange = range(min, max+1)
  print myRange
```

Every function has the following structure

- the word 'def' (for define)
- a name – a meaningful and informative name
- a pair of parentheses, possibly empty, possibly with arguments
- a colon (indicating that more is to follow)
- a body – the instructions to say what the function should do – this must be indented from the def

The range Function

Range is an interesting function that gives us all the values in the range we specify

```
>>> print range(1, 10)
>>> print range(1, 100)
```

```
>>>
>>>
>>>
>>>
>>> print range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(1,100)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
>>>
```

range

Range is an interesting function that gives us all the values in the range we specify

```
>>> print range(1, 10)
>>> print range(1, 100)
```

```
>>>
>>>
>>>
>>>
>>> print range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(1,100)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
>>>
```

Notice that it doesn't print the last number in the range – 10 or 100

range

It gives us all the integer values from the first one (inclusive) to the last one (exclusive)

That is, all the values from the first to one less than the last

There is a good reason for this, but it's tricky, so for now it's easiest just to accept it

```
>>>
>>>
>>>
>>>
>>> print range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(1,100)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
>>>
```


Calling Functions

When we call a JES function, we type its name and a pair of parentheses

We might also need to type some values within the parentheses

Calling Functions

When we define these values in the function they are described as *parameters* when we *call* the function they are called *arguments*

Parameters

```
def showRange(min, max):
  myRange = range(min, max+1)
  print myRange
```

```
>>> showRange(2,5)
```

Arguments

Arguments

`range( __, __ )`	requires two arguments, a start and end value (both integers) – also allows for an optional third argument (step size)
`requestString(__)`	requires one argument, a message (a string)
`pickAFile()`	requires no arguments

Arguments

If we call a function with the wrong number or types of arguments, JES will tell us we've made an error

The error message can be helpful, but you might have to work to understand it.

```
>>> makePicture()
The error was:makePicture() takes at least 1 argument (0 given)
Inappropriate argument type.
An attempt was made to call a function with a parameter of an invalid type. This means that you did something
such as trying to pass a string to a method that is expecting an integer.
>>>
```


Parameters

When we write a function that will take arguments we include parameters to match to each argument we need.

What type of parameters and how many parameters depends on what you are trying to do in the function (so a programmer's decision)

Parameters are part of the *signature* of the function.

Arguments and parameters

For example, when somebody wrote the `range` function, in Python they included two parameters.

Parameters

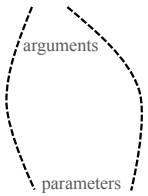
```
def showRange(min, max):
  myRange = range(min, max+1)
  print myRange
```

When I wrote my `showRange` function I decided it would be good to have 2 integer parameters. One for the `min` and one for the `max`.

Arguments and parameters

When we use (call) a function with parameters, we need to include arguments that correspond to each parameter

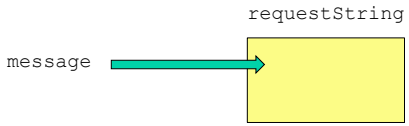
```
>>> showRange(2,5)
```




```
def showRange(min, max):
  myRange = range(min, max+1)
  print myRange
```

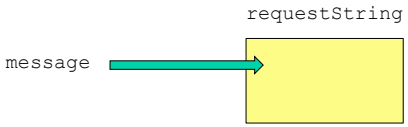

Getting Information in

Parameters are the correct way to get the information that you need into a function.



Getting Information in

Parameters are the correct way to get the information that you need into a function.

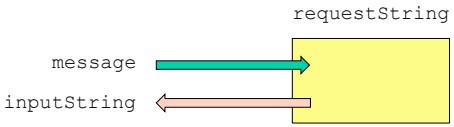


Novice programmers sometimes try to use global variables to get information in and out of a function - you should never do this!

Functions can also return something

A function like `requestString` produces a result that can be assigned to a variable

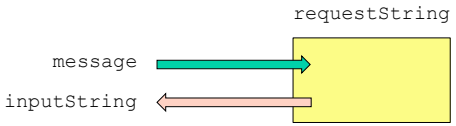
```
inputString = requestString("Enter your name")
```



Functions can also return something

A function like `requestString` produces a result that can be assigned to a variable

```
inputString = requestString("Enter your name")
```



In the terminology of functions, `requestString` *returns* a value

Functions returning things

We can get a function to return a value by including a *`return`* statement, with the value, as the last statement in the function definition.

Functions returning things

We can get a function to return a value by including a *`return`* statement, with the value, as the last statement in the function definition.

Novice programmers sometimes try to use global variables to get information in and out of a function - you should never do this! Use a return statement!

A function that returns a value

```
def sumThree(a, b, c):
  sum = a + b + c
  return sum
```

```
>>> sumThree(4, 5, 1)
```

will *return* a value of `10`

A function that returns a value

That value that is returned can then be assigned to a variable or used in other ways

```
>>> z = sumThree(5, 2, 1)
>>> print z

>>>> print 3 * sumTo(1, 2, 3)
```


Many unhappy returns

Many programmers (including the textbook) use more than one return statement.

One of our programming style requirements is to only ever use one return statement in a function.

This means there is one exit point from the function (this avoids complex jumps, or gotos, in the code and help readability and maintenance)

Functions to avoid repeated code

Novice programmers often ask “When do I need to write a function? Can’t I just have a long block of code or copy and paste the code I need?

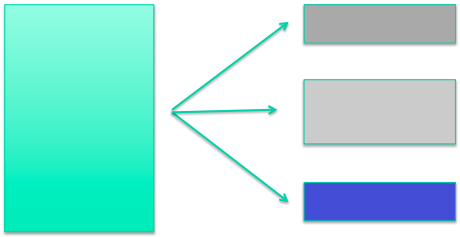
Functions to avoid repeated code

Novice programmers often ask “When do I need to write a function? Can’t I just have a long block of code or copy and paste the code I need?

Whenever you find a significant chunk of code being repeated, rewrite it as a function, with arguments for the bits that vary, and replace the repeated chunks with repeated calls to this new function

Functions to improve code quality

If you have a big difficult problem - break it into smaller easier parts – each of these parts can often be implemented as a separate function.



Functions to improve code quality

If you have a big difficult problem - break it into smaller easier parts – each of these parts can often be implemented as a separate function.

This is much easier than writing “big” functions.

Having small blocks of code that do a specific job often improves the quality of code.

Once you test your small function you know it will be a reliable piece of code. You can often reuse these useful, well-tested functions.

Function - pickAFile()

```
>>> filename=pickAFile()
```

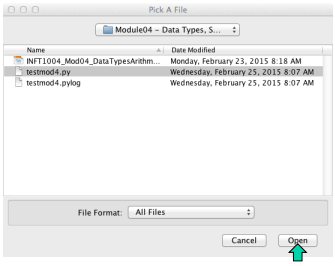
The pickAFile() function requires no arguments
The pickAFile() function returns a string

pickAFile()

```
>>> filename=pickAFile()
```

The pickAFile() function requires no arguments
The pickAFile() function returns a string

```
>>> print filename
```



The string it returns is the path to the file.

pickAFile()

```
>>> filename=pickAFile()
```

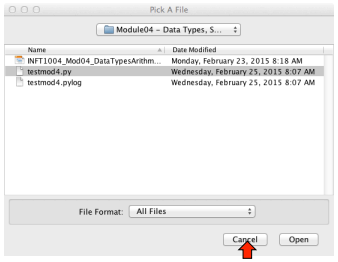
It lets you pick a file (although not always very well from external drives, USBs)

pickAFile()

```
>>> fileName=pickAFile()
```

What happens if you press the cancel button? It returns None

```
>>> print fileName
None
```



pickAFile()

```
>>> fileName=pickAFile()
```

What happens if you press the cancel button? It returns None

```
>>> print fileName
None
```

None is a constant defined in python that is often used when a function normally returns something but sometimes doesn't.

pickAFile()

We will need to be able to do different things – depending on which button the user selects

This requires "Selection" statements in the code (later)

Lets look ahead a bit

pickAFile()

```
def testPickAFile():
  # This function uses pickAFile function to select a file
  # it prints the path and name of the the file if one is selected
  # If no file is selected then it prints an appropriate message

  filename = pickAFile()

  if filename == None:
     print "No filename was selected"
  else:
     print "Selected File = " + filename


  >>> testPickAFile()
```

Mod3_2_testFunctions.py

Mod04_01_testFunctions.py

pickAFile()

```
def testPickAFile():
  # This function uses pickAFile function to select a file
  # it prints the path and name of the the file if one is selected
  # If no file is selected then it prints an appropriate message

  filename = pickAFile()

  if filename == None:
     print "No filename was selected"
  else:
     print "Selected File = " + filename


  >>> testPickAFile()
```

Notice the use of the if statement to test for "None"

the else statement can be used with an if statement to do something else

pickAFile()

```
def testPickAFile():
  # This function uses pickAFile function to select a file
  # it prints the path and name of the the file if one is selected
  # If no file is selected then it prints an appropriate message

  filename = pickAFile()

  if filename == None:
     print "No filename was selected"
  else:
     print "Selected File = " + filename
```

"=" This is the assignment operator

"==" This is a comparison operator (It is one of many used in selection)

pickAFile()

```
def testPickAFile():
  # This function uses pickAFile function to select a file
  # it prints the path and name of the the file if one is selected
  # If no file is selected then it prints an appropriate message

  filename = pickAFile()

  if filename == None:
     print "No filename was selected"
  else:
     print "Selected File = " + filename
```

"=" This is the assignment operator

"==" This is a comparison operator (It is one of many used in selection)

We will talk more about selection and iteration soon!

INFT1004

Introduction to Programming

Module 3.3
JES Media paths

Files and Media Path

Two important functions when working with files are:

getMediaPath()

setMediaPath()

Files and Media Path

Notice by default that JES looks for any files you try to open in the current media path. (It also writes there)

Type this command in the command window and you can see where it is currently looking…

```
>>> getMediaPath()
```


Files and Media Path

You can always change the media path using

```
>>> setMediaPath()
```

You would normally do this in the command window (not the program itself)

Files and Media Path

```
setMediaPath(directory)
```

directory: The directory you want to set as the media folder (optional).

Takes a directory as input. JES then will look for files in that directory unless given a full path.

You can leave out the directory. If you do, JES will open up a file chooser to let you select a directory.

Files and Media Path

```
>>> setMediaPath()
```

When working on your assignment - use this command from the command window to first select the directory where your data files are being read from and stored to

DON'T do it in the program itself

Files and Media Path

Once set you can then use the media path in your program to open a file (and it will specify the exact path)

```
image = makePicture(getMediaPath("beach.jpg"))

writePictureTo(newPicture, getMediaPath("new.png"))
```


Files and Media Path

This will be useful when working with all types of files in JES

```
filename = "girlNames.txt"

# concatenate the media path with the
# actual filename to get the full path name
fullPathName= getMediaPath(filename)

file = open(fullPathName, "r")
```

Files and Media Path

`getMediaPath(filename)`

filename: the name of the file you want (optional)

returns: the complete path to the file specified

This function builds the whole path to the file you specify, as long as you've already used `setMediaPath()` to pick out the place where you keep your media.

If no filename is given, only the MediaPath will be returned.

Programs in the book

Also when working with these programs, note the benefit of using the `setMediaPath()` function of JES

It tells JES where to start looking for files (so you don't have to supply the whole path)

Filenames in the book

We can use `getMediaPath()` as part of a filename in a function

(remember to set the media path first)

You will be doing this in the assignment

Mac vs Windows

Be aware there are some slight variations in the way file paths are specified in Mac and Windows

"`C:\ip-book\mediasources\beach.jpg`" (Windows)

"`Desktop/ip-book/mediasources/beach.jpg`" (Mac)

But this will only be a problem if you hard code path names – so another good reason to use getMediaPath and setMediaPath

What to do this week

- [ ] Do the Quiz for Week 3
- [ ] Check the Tutorial solution from Week 2 (if you need to)
- [ ] Start on the Week 3 tutorials (bring your problems to class)
- [ ] Keep reading the textbook (Ch 1, 2, 3, 4)

117