	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 - SEMESTER 1 - 2017 LECTURE TOPICS

Practical Test in Tutorial
(Week 5)

Assignment set
(12 April, 8:00am)

Lecture Topics and Lab topics are the same for each week

INFT1004

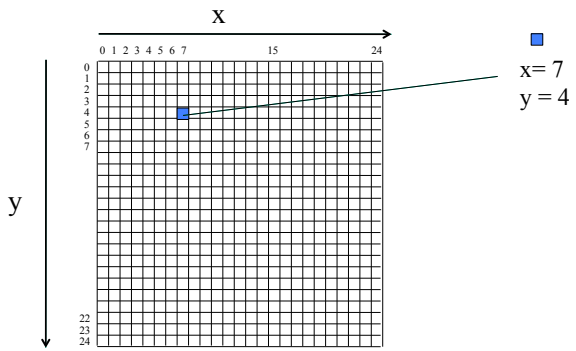
Introduction to Programming

Module 4.1
Arrays and Pixels

Guzdial & Ericson - Third Edition – chapters 2, 3
Guzdial & Ericson - Fourth (Global) Edition – chapters 2 and 4

Revision
getPixel()

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



Revision
working with individual pixels

We can name individual pixels in the picture, get there color and set there color

```
pixel1 = getPixel(picture1, 32, 32)
pixel2 = getPixel(picture1, 33, 33)
colorPixel1 = getColor(pixel1)
setColor(pixel1, yellow)
setColor(pixel2, red)
explore(pic1)
```


Revision
Pixel color

Colours have a red, green and blue channel

```
pixel3 = getPixel(picture1, 12, 20)

setRed(pixel3, 0)       # no red
setGreen(pixel3, 100)   # value in [0,255]
setBlue(pixel3, 255)    # max blue

redPart = getRed(pixel3)
greenPart = getGreen(pixel3)
bluePart = getBlue(pixel3)
```


Revision
Color in JES

You can make your own colours or pick one

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

myColor = pickAColor()
```

Iteration

One of the most powerful aspects of programming is the ability to tell the program to repeat the same thing over and over, with minor variations

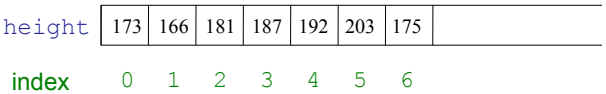
This is called *iteration*

We're going to use iteration a lot when processing pictures

But first we need to know about *sequences*

Sequence / List

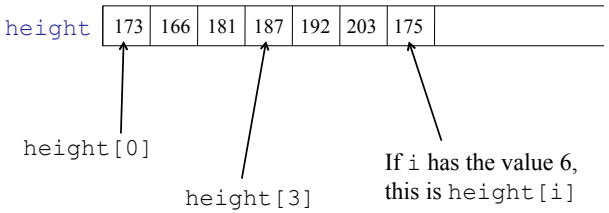
An sequence is a collection of items
all of which have the same name
but each of which has a different index, an identifying number



Note: the first element has index 0,
the second has index 1,

Sequence / List

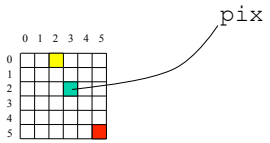
An array is a collection of items
all of which have the same name
but each of which has a different index, an identifying number



getPixel() gives us a pixel

getPixel() gives us a single pixel from a picture

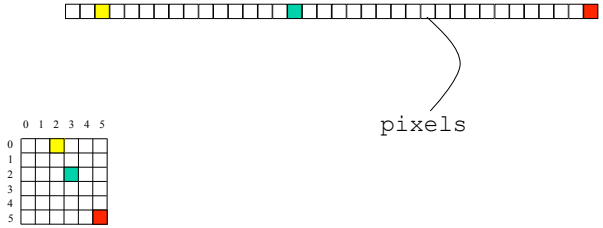
```
pix = getPixel(myPic, 3, 2)
```



getPixels() returns a sequence

getPixels() gives us a sequence containing every pixel from the picture!

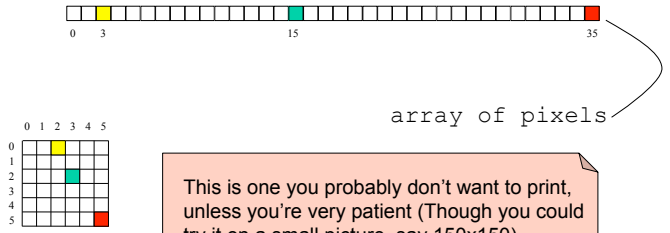
```
pixels = getPixels(myPic)
```



getPixels() returns a sequence

getPixels() gives us a sequence containing every pixel from the picture!

```
pixels = getPixels(myPic)
```

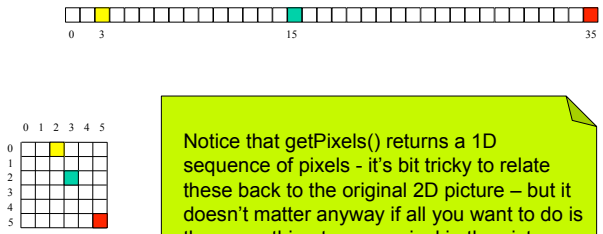


This is one you probably don't want to print, unless you're very patient (Though you could try it on a small picture, say 150x150)

getPixels() returns a sequence

getPixels() gives us a sequence containing every pixel from the picture!

```
pixels = getPixels(myPic)
```

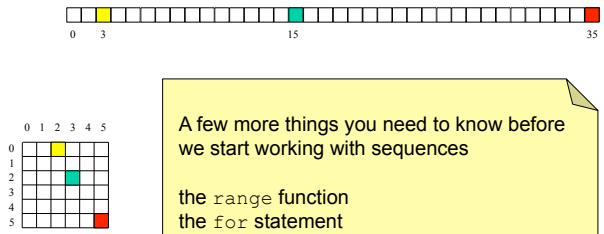


Notice that getPixels() returns a 1D sequence of pixels - it's bit tricky to relate these back to the original 2D picture – but it doesn't matter anyway if all you want to do is the same thing to every pixel in the picture

getPixels() returns a sequence

getPixels() gives us a sequence containing every pixel from the picture!

```
pixels = getPixels(myPic)
```



A few more things you need to know before we start working with sequences

the `range` function
the `for` statement

Revision

range function

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.

You can also use three arguments with the range function, Try…

```
>>> range(10, 3, -1)
>>> range(0, 21, 2)
```


Revision

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)
```

Notice that it doesn't print the last number in the range – 10 or 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]
>>>
```


The *for* statement and getPixels

The `for` statement is a powerful iteration statement – you might want to process every pixel in a picture (Here I halve the amount of red in picture)

```
for pixel in getPixels(picture):
  value = getRed(pixel)
  setRed(pixel, value * 0.5)
```

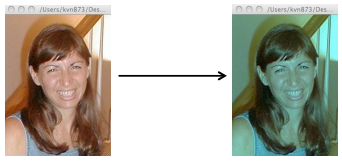
Mod4_1_ArraysAndPixels.py

The *for* statement and getPixels

The `for` statement is a powerful iteration statement – you might want to process every pixel in a picture (Here I halve the amount of red in picture

```
for pixel in getPixels(picture):
  value = getRed(pixel)
  setRed(pixel, value * 0.5)
```

This gives every pixel in the picture a new red value that's half the red value it had before (It doesn't alter the values in the green or blue channels)



barbara.jpg

Mod4_1_ArraysAndPixels.py

What the *for* statement does

The *for* statement basically does its body for every single one of the values in the collection

Note that the variable `pixel` is assigned a different pixel from the array every time through the loop

```
for pixel in getPixels(picture):
  value = getRed(pixel)
  setRed(pixel, value * 0.5)
```


What the *for* statement does

That is . . .

it assigns the variable to the first value and does the body

```
for pixel in getPixels(picture):
  value = getRed(pixel)
  setRed(pixel, value * 0.5)
```


What the *for* statement does

That is . . .

it assigns the variable to the first value and does the body

then it assigns the variable to the next value and does the body

```
for pixel in getPixels(picture):
  value = getRed(pixel)
  setRed(pixel, value * 0.5)
```


What the *for* statement does

That is . . .

it assigns the variable to the first value and does the body

then it assigns the variable to the next value and does the body

then it assigns the variable to the next value and does the body

```
for pixel in getPixels(picture):
  value = getRed(pixel)
  setRed(pixel, value * 0.5)
```

What the *for* statement does

That is . . .

it assigns the variable to the first value and does the body

then it assigns the variable to the next value and does the body

then it assigns the variable to the next value and does the body

and so on until it's done the body for every value in the collection

```
for pixel in getPixels(picture):
  value = getRed(pixel)
  setRed(pixel, value * 0.5)
```


The *for* statement and getPixels

This code does exactly the same thing – make sure you understand why!

```
for myPixel in getPixels(picture):
  value = getRed(myPixel)
  setRed(myPixel , value * 0.5)
```

```
arrayPixels = getPixels(picture)
numPixels = len(arrayPixels)

for i in range(0, numPixels):
  myPixel = arrayPixels[i]
  value = getRed(myPixel )
  setRed(myPixel , value * 0.5)
```


The *for* statement and getPixels

Th
ur

Which is best ?
In this example the first way is probably a bit simpler – but the second approach is useful if you don't want to process every pixel in the picture.

```
arrayPixels = getPixels(picture)
numPixels = len(arrayPixels)

for i in range(0, numPixels):
  myPixel = arrayPixels[i]
  value = getRed(myPixel )
  setRed(myPixel , value * 0.5)
```


Iteration within the picture

Le's try altering just some of the pixels; try this . . .

```
def cyanBlock(picture):
  copyPicture = duplicatePicture(picture)
  pixelArray = getPixels(copyPicture)
  numPixels = len(pixelArray)

  #process only the first half of the pixels
  for i in range(0, numPixels/2):
    setColor(pixelArray[i], cyan)

  return copyPicture
```

Iteration within the picture

```
def testCyanBlock():
    ## tests the cyanBlock function

    myFile = pickAFile()
    myPicture = makePicture(myFile)
    show(myPicture)

    cyanPicture = cyanBlock(myPicture)
    repaint(cyanPicture)
```


Mod4_1_ArraysAndPixels.py

Explore, show, and repaint

`explore()` is useful when we want to examine individual pixels of a picture, but it can be annoying that it opens a new window every time we call it - Instead try

```
>>> show(myPic)
>>> repaint(myPic)
```


Explore, show, and repaint

`explore()` is useful when we want to examine individual pixels of a picture, but it can be annoying that it opens a new window every time we call it - Instead try

```
>>> show(myPic)
>>> repaint(myPic)
```

Even `show(myPic)` has a strange quirk: it sometimes seems to show a previous version of the picture.

So maybe we should just stick with `repaint()` unless we particularly want to use the features of `explore()`

Saving altered pictures

Want to save a copy of a picture you've altered?

```
writePictureTo(myPic, "newPic.jpg")
```

This will save the new file in the same directory that you picked the old file from

Saving altered pictures

Want to save a copy of a picture you've altered?

```
writePictureTo(myPic, "newPic.jpg")
```

This will save the new file in the same directory that you picked the old file from

Or you can use the media path to help

```
writePictureTo(myPic, getMediaPath("newPic.jpg"))
```


Saving altered pictures

Want ed?

wr pg")

This v ry that you picke

Or you can use the media path to help

```
writePictureTo(myPic, getMediaPath("newPic.jpg"))
```

You should find this quite useful for the assignment - when you need to save pictures a file.

Lets use some more iteration

Lets solve some problems that need us to process all the pixels in the image. We can use getPixels for this!

```
myPixels = getPixels(myPic)

for pixel in myPixels :
  <do something to pixel>
```

Examples

1. Halving a color
2. Doubling a color
3. Adjusting a color
4. Creating a negative of an image
5. Creating a greyscale of a color image

Doubling a colour

```
def doubleGreen(picture):
  pixels = getPixels(picture)
  for pixel in pixels:
    greenValue = getGreen(pixel)
    setGreen(pixel, greenValue * 2)
```

This gives every pixel in the picture a new green value that's double the green value it had before (It doesn't alter the values in the red or blue channels)

barbara.jpg  doubleGreen → 

Doubling a colour

Remember a color has a maximum value of 255

200 * 2 = 400 ???

Doubling a colour

Remember a color has a maximum value of 255

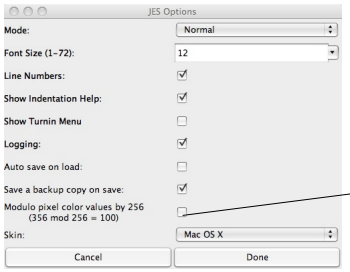
200 * 2 = 400 ??? 400 modulo 256 = 145

This may have unexpected results – you may want to keep the maximum value of 255

Doubling a colour

Remember a color has a maximum value of 255

200 * 2 = 400 ??? 400 modulo 256 = 145



What really happens actually depends on how the options are set - either to wrap colours (modulo) or not

– see Edit/Options & modulo.

Adjusting a colour

```
def adjustBlue(picture, amount):
  newPicture = duplicatePicture(picture)
  pixels = getPixels(newPicture )
  for pixel in pixels:
    blueValue = getBlue(pixel)
    setBlue(pixel, blueValue * amount)
  return newPicture
```

This gives every pixel in the picture a new blue value that's *amount* times the blue value it had before

It can be used to increase (amount > 1) or decrease (amount < 1) blue values, depending on the value of amount (and how the Modulo option is set)

Always be aware of types

If you're to understand the programs in the lectures and the book, one of the things you must stay on top of is the type of each object.

Always be aware of types

In the methods we've just looked at, the variable…

`picture` is a picture object (not, for example, a file object)

`pixels` is a sequence, the sequence of all the pixels in the picture

`pixel` is an individual pixel; but each time through the loop it's a different pixel

`greenValue`, `redValue`, and `blueValue` are integers

Always be aware of types

In the methods we've just looked at, the variable…

`pict`…le, a file ob…

`pixe`…e pixels…

`pixe`…

through the loop it's a different pixel

`greenValue`, `redValue`, and `blueValue` are integers

Always be sure you know the type of every item!

Always be aware of types

`greenValue`, `redValue`, and `blueValue` are integers

Note that it is tempting to use variable names like

`red, blue, green`.

But remember these are already defined as colors in JES. If you assign them to be something else then you may get confused when JES doesn't recognise them as colors anymore – so don't!

Negative

Some of you might be familiar with photographic negatives, although they don't exist in digital photography

The negative has colours the exact opposite of the 'normal' picture

Negative

In RGB terms, the opposite of a colour is 255 minus that colour in each of the three channels

To make a negative,
- set the red to 255 minus what it was
- set the green to 255 minus what it was
- set the blue to 255 minus what it was

```
>>> negRed = 255 - currentRed
>>> negGreen = 255 - currentGreen
>>> negBlue = 255 - currentBlue
```


Negative

In RGB terms, the opposite of a colour is 255 minus that colour in each of the three channels

To make a negative,
- set the red to 255 minus what it was
- set the green to 255 minus what it was
- set the blue to 255 minus what it was

```
>>> negRed = 255 - currentRed
>>> negGreen = 255 - currentGreen
>>> negBlue = 255 - currentBlue
```

You will do this in this weeks tut

Negative

Unlike the doubling, this one won't have colour values wrapping around. Why not?

To make a negative,
- set the red to 255 minus what it was
- set the green to 255 minus what it was
- set the blue to 255 minus what it was

```
>>> negRed = 255 - currentRed
>>> negGreen = 255 - currentGreen
>>> negBlue = 255 - currentBlue
```

You will do this in this weeks tut

Greyscale

In RGB terms, black, grey, and white all have one thing in common – equal red, green, and blue values

red = 0 green = 0 blue = 0	red = 60 green = 60 blue = 60	red = 120 green = 120 blue = 120	red = 180 green = 180 blue = 180	red = 255 green = 255 blue = 255

Greyscale

To turn a colour picture into a greyscale picture, we need to set red, green, and blue to the average of their original values (add up red, green, blue and divide by 3)

```
greyVal = (getRed(pixel) + getGreen(pixel) + getBlue(pixel))/3
setRed(pixel, greyVal)
setGreen(pixel, greyVal)
setBlue(pixel, greyVal)
```

All this, of course, in a loop in a function so that each pixel in the image is changed.

INFT1004

Introduction to Programming

Module 4.2
Iteration – For Loop

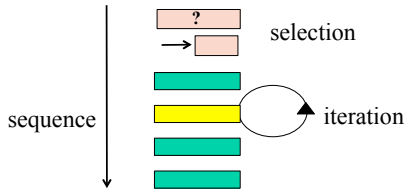
Guzdial & Ericson - Third Edition – chapters 3, 4 and 5
Guzdial & Ericson - Fourth (Global) Edition – chapters 4, 5 and 6

Sequence, selection, iteration

Programming has three essential building blocks

We've already met sequence

Iteration determines which code to execute depending on specified conditions



Types of Loops

While loop	Tests some condition - If the condition is true it executes some statements Repeats until the test condition is false
For loop	Repeats some statements a predetermined number of times
Nested loop	One loop inside another loop

Types of Loops

While loop	Tests some condition - If the condition is true it executes some statements Repeats until the test condition is false

I found the while loop quite useful for the assignment as you are dealing with lists and you generally don't know how many things are in the list.

Types of Loops

We will focus mostly on the for loop in this course.

(Mostly we will know how many times we need to loop)

For loop	Repeats some statements a predetermined number of times
Nested loop	One loop inside another loop

Types of Loops

We will focus mostly on the for loop in this course.

(Mostly we will know how many times we need to loop)

We will use the nested loop quite a lot when processing pictures – soon)

Nested loop	One loop inside another loop

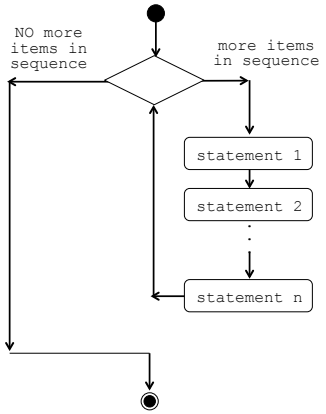
Types of Loops

While loop	Tests some condition - If the condition is true it executes some statements Repeats until the test condition is false
For loop	Repeats some statements a predetermined number of times
Nested loop	One loop inside another loop

For Loop

Repeats some statements a predetermined number of times

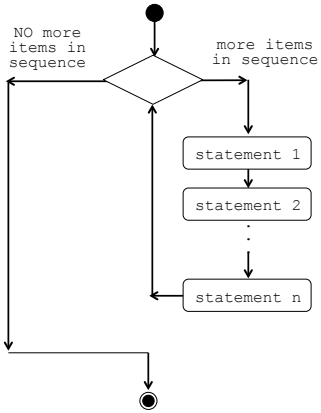
We tend to use this when we know how many times we need to loop (do it 4 times, process every element in a list)



For Loop – using range

We can use the range function to help us loop a set number of times

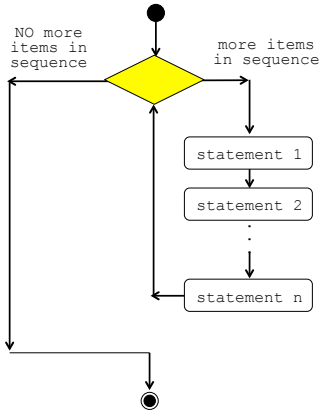
We tend to use this when we know how many times we need to loop (do it 4 times, process every element in a list)



For Loop – using range

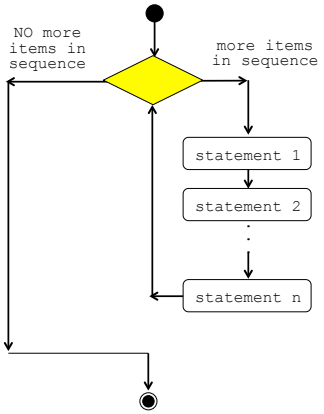
```
for var in <sequence> :
    statement(s)
```

```
def testForRange():
    for count in range(1,5):
        print(count)
```



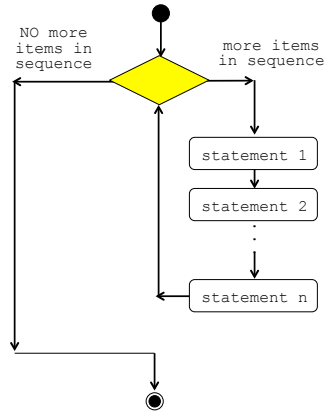
For Loop – using range

```
for var in <sequence> :
      statement(s)

def testForRange():
   for count in range(1,5):
      print(count)
```



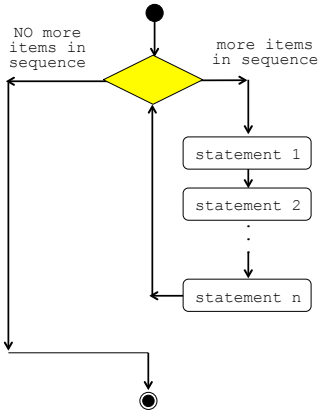
For Loop – using range

```
for var in <sequence> :
      statement(s)

def testForRange():
   for count in range(1,5):
      print(count)
```



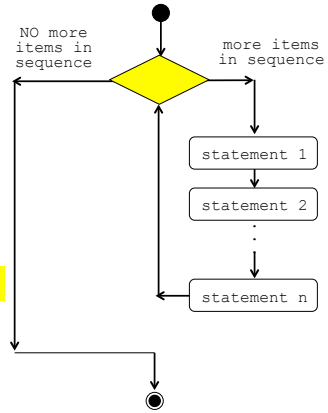
For Loop – using range

```
for var in <sequence> :
      statement(s)

def testForRange():
   for count in range(1,5):
      print(count)

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



For Loop – using range

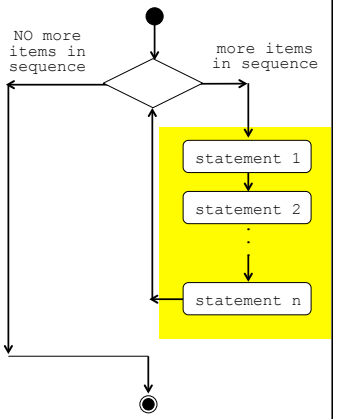
```
for var in <sequence> :
      statement(s)

def testForRange():
   for count in range(1,5):
      print(count)
```



For Loop – using range

```
for var in <sequence> :
      statement(s)
```

```
def testForRange():
    for count in range(1,5):
        print(count)
```



For Loop – example code

```
def testForRange():
    # This function uses for statement to print
    # all the numbers from 1 to 4
    for count in range(1,5):
      print(count)
```

Mod4_2_testIterationFor.py

For Loop – Desk Check

```
def testForRange():
    # This function uses for statement to print
    # allthe numbers from 1 to 4
    for count in range(1,5):
      print(count)
```

Let's check this using some desk checking

Mod4_2_testIterationFor.py

For Loop – Desk Check

count	range(1,5)	print
1	[1,2,3,4]	

⇨ `for count in range(1,5):`
`    print(count)`

For Loop – Desk Check

count	range(1,5)	print
1	[1,2,3,4]	
		1

```
for count in range(1,5):
    print(count)
```


For Loop – Desk Check

count	range(1,5)	print
1	[1,2,3,4]	
		1
2	[1,2,3,4]	

```
for count in range(1,5):
    print(count)
```


For Loop – Desk Check

count	range(1,5)	print
1	[1,2,3,4]	
		1
2	[1,2,3,4]	
		2

```
for count in range(1,5):
    print(count)
```


For Loop – Desk Check

count	range(1,5)	print
1	[1,2,3,4]	
		1
2	[1,2,3,4]	
		2
3	[1,2,3,4]	

```
for count in range(1,5):
    print(count)
```

For Loop – Desk Check

count	range(1,5)	print
1	[1,2,3,4]	
		1
2	[1,2,3,4]	
		2
3	[1,2,3,4]	
		3

```
for count in range(1,5):
    print(count)
```


For Loop – Desk Check

count	range(1,5)	print
1	[1,2,3,4]	
		1
2	[1,2,3,4]	
		2
3	[1,2,3,4]	
		3
4	[1,2,3,4]	

```
for count in range(1,5):
    print(count)
```


For Loop – Desk Check

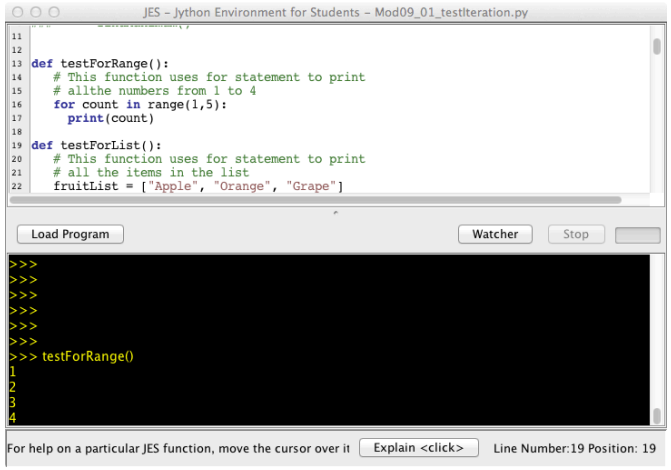
count	range(1,5)	print
1	[1,2,3,4]	
		1
2	[1,2,3,4]	
		2
3	[1,2,3,4]	
		3
4	[1,2,3,4]	
		4

```
for count in range(1,5):
    print(count)
```


For Loop – Desk Check

count	range(1,5)	print
1	[1,2,3,4]	
		1
2	[1,2,3,4]	
		2
3	[1,2,3,4]	
		3
4	[1,2,3,4]	
		4

```
for count in range(1,5):
    print(count)
```

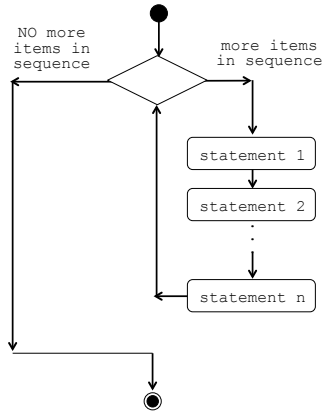


Mod4_2_testIterationFor.py

For Loop – with a List

We can use this to loop through all the items in a list. (Handy for your assignment)

We tend to use this when we know how many times we need to loop (do it 4 times, process every element in a list)



For Loop – with a List

```
for var in <sequence> :
      statement(s)


def testForList():
   fruitList = ["Apple", "Orange", "Grape"]

   for fruit in fruitList:
      print(fruit)
```


For Loop – with a List

```
for var in <sequence> :
      statement(s)


def testForList():
   fruitList = ["Apple", "Orange", "Grape"]

   for fruit in fruitList:
      print(fruit)
```

For Loop – with a List

```
for var in <sequence> :
     statement(s)


def testForList():
   fruitList = ["Apple", "Orange", "Grape"]

   for fruit in fruitList:
      print(fruit)
```


For Loop – with a List

```
for var in <sequence> :
     statement(s)


def testForList():
   fruitList = ["Apple", "Orange", "Grape"]

   for fruit in fruitList:
      print(fruit)
```

Mod 4.2 Iteration – For Loop

For Loop – with a List

```
for var in <sequence> :
     statement(s)


def testForList():
   fruitList = ["Apple", "Orange", "Grape"]

   for fruit in fruitList:
      print(fruit)
```

Mod 4.2 Iteration – For Loop

For Loop – with a List

```
for var in <sequence> :
     statement(s)


def testForList():
   fruitList = ["Apple", "Orange", "Grape"]

   for fruit in fruitList:
      print(fruit)
```

Mod 4.2 Iteration – For Loop

For Loop – example code

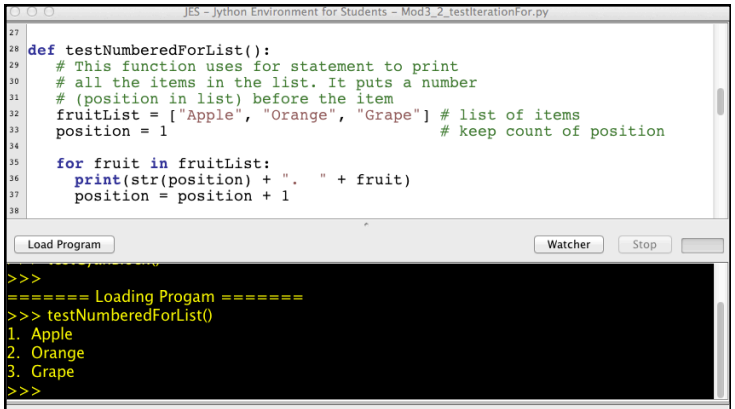
```
def testForList():
    # This function uses for statement to print
    # all the items in the list
    fruitList = ["Apple", "Orange", "Grape"]

    for fruit in fruitList:
      print(fruit)
```

Mod4_2_testIterationFor.py

For Loop – example code

```
def testNumberedForList():
    # This function uses for statement to print
    # all the items in the list. It puts a number
    # (position in list) before the item

    fruitList = ["Apple", "Orange", "Grape"]
    position = 1

    for fruit in fruitList:
       print(str(position) + ".  " + fruit)
       position = position + 1
```

Mod4_2_testIterationFor.py



Mod4_2_testIterationFor.py

For Loop – Maximum

Let's say you were asked to find the maximum number in a list of numbers.

I might first write a function that tests my code

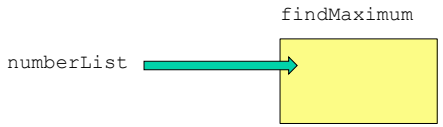
```
def testMaximumFunction():
    #a list of numbers to test
    listNumbers = [89.3, 42.6, 91.7, 28.0]

    #I still need to write the findMaximum function
    myMax = findMaximum(listNumbers)

    #print the result to check it works
    # it should be 91.7 in this list
    print("Maximum = " + str(myMax))
```

Mod4_2_testIterationFor.py

For Loop – Maximum

I still need to write the findMaximum function

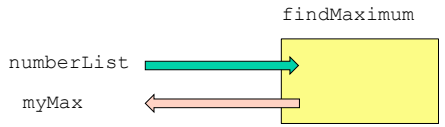
I decide to pass in a list (of numbers) – so this will be a parameter



For Loop – Maximum

I still need to write the findMaximum function

I decide to pass in a list (of numbers) – so this will be a parameter



I decide to return the maximum number in the list

For Loop – Maximum

Notice when I call findMaximum – I pass in a list of numbers and I save the returned value

```
def testMaximumFunction():
    #a list of numbers to test
    listNumbers = [89.3, 42.6, 91.7, 28.0]

    #I still need to write the findMaximum function
    myMax = findMaximum(listNumbers)

    #print the result to check it works
    # it should be 91.7 in this list
    print("Maximum = " + str(myMax))
```

Mod4_2_testIterationFor.py

For Loop – findMaximum

```
def findMaximum(numberList):
```

```
    return maximum
```

Notice when I define findMaximum – I use a parameter for the list of numbers and I will need to return a value

For Loop – findMaximum

```
def findMaximum(numberList):
    #A function that returns the maximum
    #number in the list
    #assumes it is a list of numbers

    maximum = numberList[0] #use first element as max

    for number in numberList:
       if number > maximum :
          maximum = number

    return maximum
```


Mod4_2_testIterationFor.py

INFT1004

Introduction to Programming

Module 4.3
Complex Types and References

Revision
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
```


Revision
Types

Python doesn't understand the English meaning of a name. It doesn't tell the type of an item by its name.

*Python is NOT a **strongly typed** language*

Revision
Types

Python doesn't understand the English meaning of a name. It doesn't tell the type of an item by its name.

It decides the type by what's **assigned** to it.

If we typed `aPixel= 3.13`

`aPixel` would be a float (because 3.13 is a float)

*Python is NOT a **strongly typed** language*

Revision
Types

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

`Pixel`, `Picture`, `Color` are all examples of more complex types (classes)

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

Revision
Types

Type is important to the computer as it has to be able to store different types in memory using only zeroes and ones.

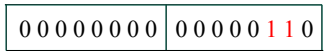
Everything in a programming environment has a type.

The type determines how the thing will be stored (coded) in memory.

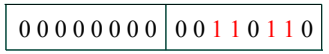
Revision
Types

Compare the integer 6 and the string "6"

integer 6 – stored as a binary number



string "6" – stored in ASCII code



Simple versus Complex Types

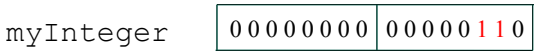
In python you will also find another key difference between simple types and classes

Variables of simple types store the value of the variable directly in a memory location

Variables of complex type store a reference (address) to the location in memory where the actual information is then stored.

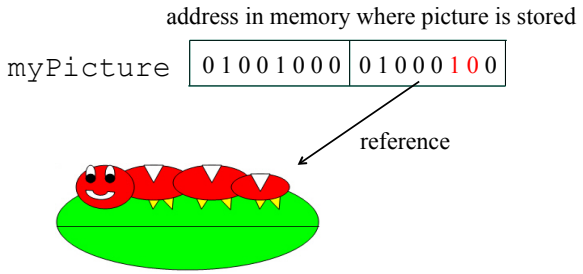
Simple versus Complex Types

Variables of simple types store the value of the variable directly in a memory location



Simple versus Complex Types

Variables of complex type store a reference (address) to the location in memory where the actual information is stored.



Assignment and References

Bug / Feature Warning: Python's use of references will mean that the assignment operator works very differently with complex types!

It is also quite different from most other languages eg. C++, Java, C#, VB, …

Revision
Assignment Statement

The '=' symbol indicates an *assignment statement*.

It takes the value on the right and assigns it to the name (variable) on its left

```
count = 5
sum = sum + number
name = "Keith"
weight = 13.57
```


Revision
Assignment Statement

Variables can be of many different types

Some are simple types, eg int, float, string - if you print one of these, you get its value

int
string
float

```
month = 10
name = "Keith"
height = 193.57

myPicture = makePicture(myPictureFile)
```


Variables have types

Variables can be of many different types

Some are simple types, eg int, float, string - if you print one of these, you get its value

Others are various types of object, eg File, Picture, Pixel - if you print one of these, you get its description

int
string
float
picture

```
month = 10
name = "Keith"
height = 193.57

myPicture = makePicture(myPictureFile)
```


Simple types, value copying

I've said the assignment statement assigns the value of what's on the right to the variable on the left

This is actually a simplification!

```
month = birthMonth
name = myName
height = playersHeight

myPicture = myFace
```

Simple types, value copying

If what's on the right is a simple type, its value is simply given to the variable on the left

They remain as two distinct quantities, each with the same value

```
month = birthMonth
name = myName
height = playersHeight

myPicture = myFace
```


Simple types, value copying

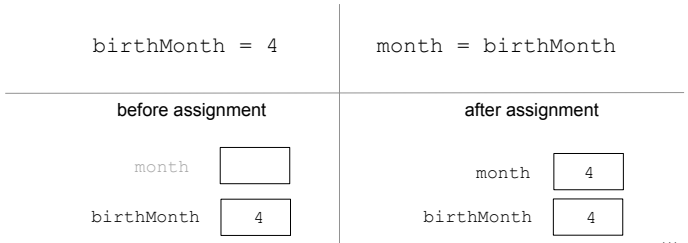
If what's on the right is a simple type, its value is simply given to the variable on the left

They remain as two distinct quantities, each with the same value

Simple types, value copying

If what's on the right is a simple type, its value is simply given to the variable on the left

They remain as two distinct quantities, each with the same value



Object types, reference copying

If what's on the right is an object type, the variable on the left becomes *a reference* (or *another name*) for the object on the right

```
month = birthMonth
name = myName
height = playersHeight

myPicture = myFace
```

Object types, reference copying

If what's on the right is an object type, the variable on the left becomes *a reference* (or *another name*) for the object on the right

There's still just a single object, which can be referred to in two different ways: it has two *references*
This distinction can be very important!

`myFace = makePicture(face)`	`myPicture = myFace`
before assignment	after assignment
`myFace` → 	`myFace` → 
`myPicture`	`myPicture` →

Object types, reference copying

```
pictureFile = pickAFile()
picture1 = makePicture(pictureFile)
picture2 = picture1
```

If you change something in `picture2` it will also change `picture1` as well

(Because they are the exact same picture – in exactly the same place in memory)

Mod4_3_ValueVersusReference.py - testValueReferenceCopying()

Simple types, value copying

```
# Value copying: the 'copy' gets the value of the original,
# but they remain different things
# Value copying happens with simple data types

a = 3
b = a       # copy value of a into b
b = 5       # change the value of b (a is not changed)
print a
print b     # b and a are different things
```

Mod4_3_ValueVersusReference.py - testValueReferenceCopying()

Object types, reference copying

```
pictureFile = pickAFile()
picture1 = makePicture(pictureFile)
show(picture1)

# Objects like pictures are copied by reference;
# picture1 & picture1 are now exactly the same object
picture2 = picture1

# pixel1 is a pixel of picture1 (and therefore picture2)
pixel1 = getPixel(picture1, 0, 0)
setColor(pixel1, green)     #change pixel in picture1
repaint(picture2)           #both pictures are changed
```

Mod4_3_ValueVersusReference.py - testValueReferenceCopying()

Note: Copy constructor

By contrast, in languages like C++ this type of assignment normally invokes the copy constructor

Thus making a copy of the object and so you get 2 different objects

(Normally in other languages objects and simple types will work the same with assignment)

(but not in python)

Side Effects – Object Arguments

Another consequence of the way python uses references with object types is that when an object is used as an argument to the function, it is passed as a reference.

This can result in a "side effect" – changing the argument inside the function will change it forever – even when you leave the function

This is not really considered a good programming practice.

Note: Copy constructor

Again in languages like C++ using an object as an argument normally invokes the copy constructor

Thus making a copy of the object, passing this copy as the argument. Any changes to the argument inside the function does not have side effects.

(but not in python)

Note: Copy constructor

Again in lang
argument nor

Thus making
as the argument. Any changes to the argument
inside the function does not have side effects.

(but not in python)

In terms of engineering, side effects are not a good thing – you should not really be using them !

Side Effects – Example

```
def drawRedSquare(picture):
    ### This function draws a red square
    ### of pixels on the picture
    ###
    pixel1 = getPixel(picture, 10, 10)
    setColor(pixel1 , red)
    pixel2 = getPixel(picture, 11, 10)
    setColor(pixel2 , red)
    pixel3 = getPixel(picture, 10, 11)
    setColor(pixel3 , red)
    pixel4 = getPixel(picture, 11, 11)
    setColor(pixel4 , red)
```


Side Effects – Example

```
def testSideEffect():
    ### This function demonstrates
    ### side effects in python
    pictureFile = pickAFile()
    myPicture = makePicture(pictureFile)
    explore(myPicture)

    ### this function has a side effect
    ### It changes the argument
    ### The explore is used to test the changes
    drawRedSquare(myPicture)
    explore(myPicture)
```


No Side Effects – Example

```
def copyDrawRedSquare(picture):
    #copy the original picture
    resultPicture = duplicatePicture(picture)

    pixel1 = getPixel(resultPicture, 10, 10)
    setColor(pixel1, red)
    pixel2 = getPixel(resultPicture, 11, 10)
    setColor(pixel2, red)
    pixel3 = getPixel(resultPicture, 10, 11)
    setColor(pixel3, red)
    pixel4 = getPixel(resultPicture, 11, 11)
    setColor(pixel4, red)

    return resultPicture
```


No Side Effects – Example

```
def testNoSideEffect():
    ### This function demonstrates
    ### how to avoid side effects in python
    pictureFile = pickAFile()
    myPicture = makePicture(pictureFile)
    explore(myPicture)

    ### this function has no side effects
    ### the argument remains unchanged
    ### the function returns a changed copy of the argument
    newPicture =  drawRedSquareFix(myPicture)
    explore(myPicture)   #no red square on original
    explore(newPicture)  #copy of picture with a red square
```

In terms of engineering, side effects are not a good thing – you should not really be using them !

What to do this week

- [ ] Do the Quiz for Week 4
- [ ] Check the Tutorial solution from Week 3 (if you need to)
- [ ] Start on the Week 4 tutorials (bring your problems to class)
- [ ] Keep reading the textbook (Ch 1, 2, 3, 4, 5)