	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 1.4
Arithmetic

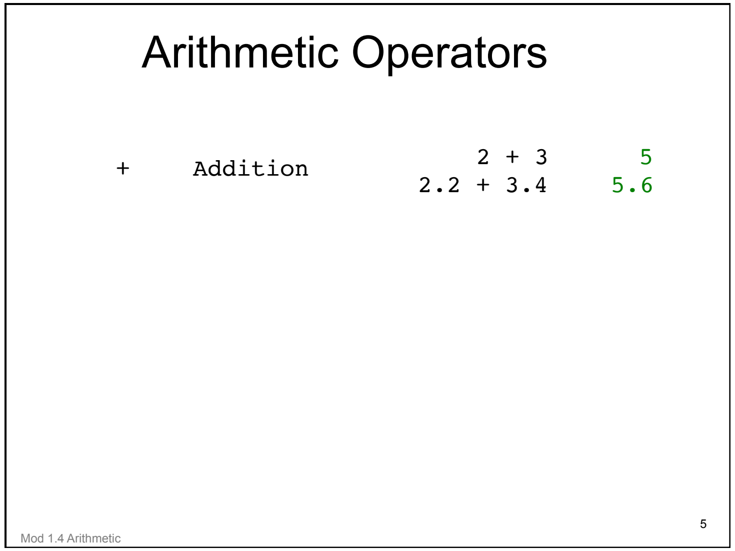
Computers do Arithmetic

Computers were designed to do arithmetic.

As a programmer it is hard to avoid some kind of mathematical thinking in solving problems.

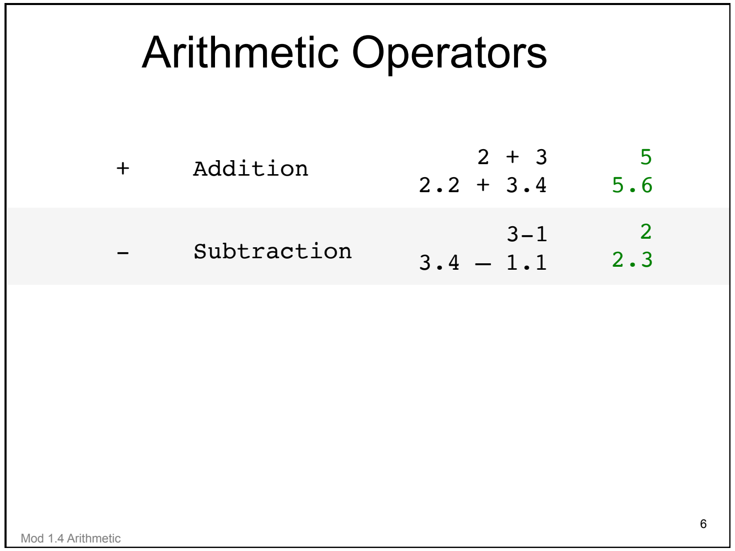
In this course we will keep things rather simple.

But it cannot be avoided completely.



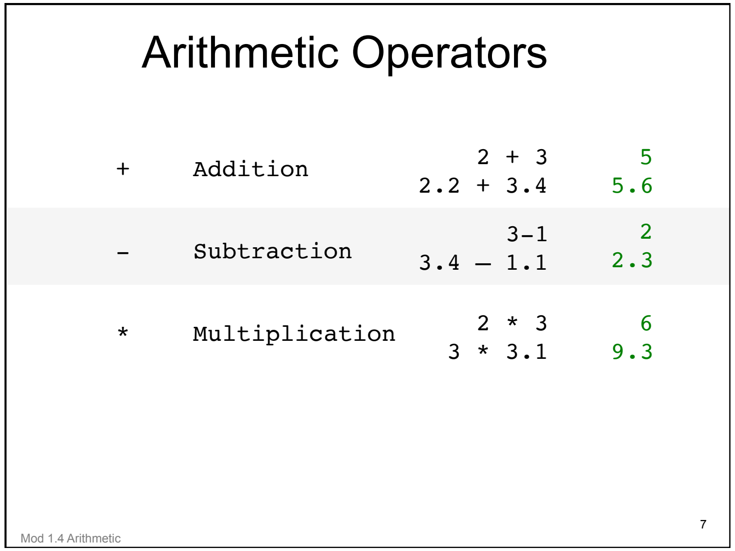
Arithmetic Operators

+	Addition	2 + 3 2.2 + 3.4	5 5.6



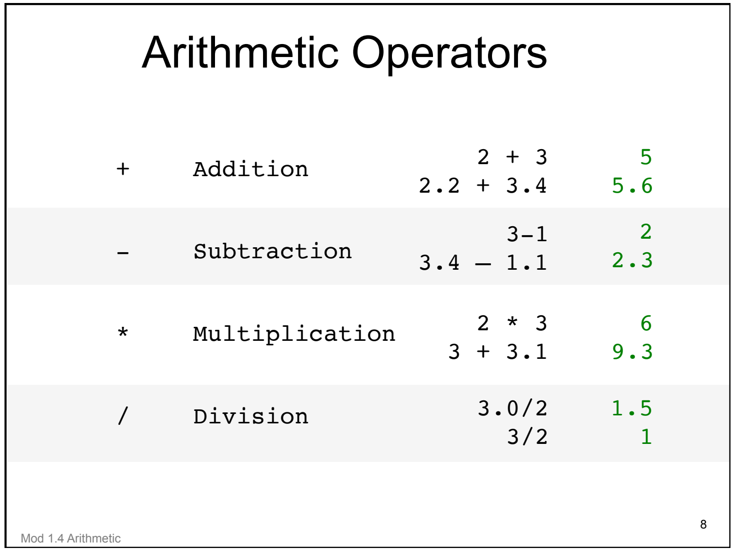
Arithmetic Operators

+	Addition	2 + 3 2.2 + 3.4	5 5.6
-	Subtraction	3-1 3.4 – 1.1	2 2.3



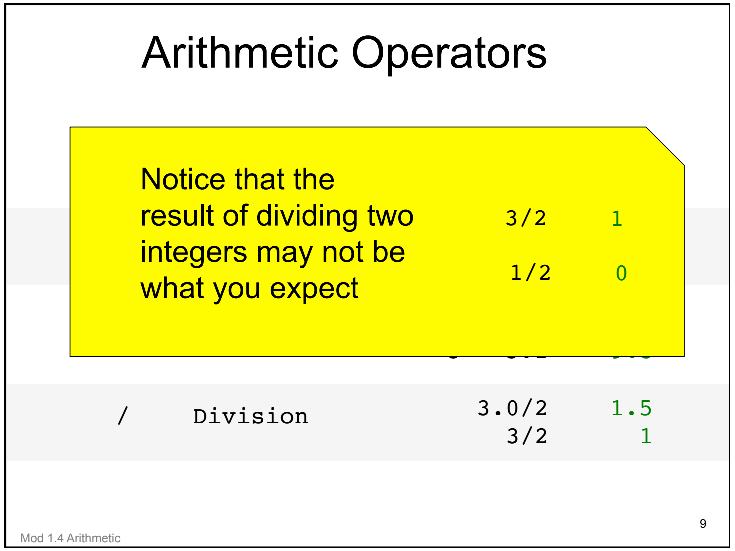
Arithmetic Operators

+	Addition	2 + 3 2.2 + 3.4	5 5.6
-	Subtraction	3-1 3.4 – 1.1	2 2.3
*	Multiplication	2 * 3 3 * 3.1	6 9.3



Arithmetic Operators

+	Addition	2 + 3 2.2 + 3.4	5 5.6
-	Subtraction	3-1 3.4 – 1.1	2 2.3
*	Multiplication	2 * 3 3 + 3.1	6 9.3
/	Division	3.0/2 3/2	1.5 1

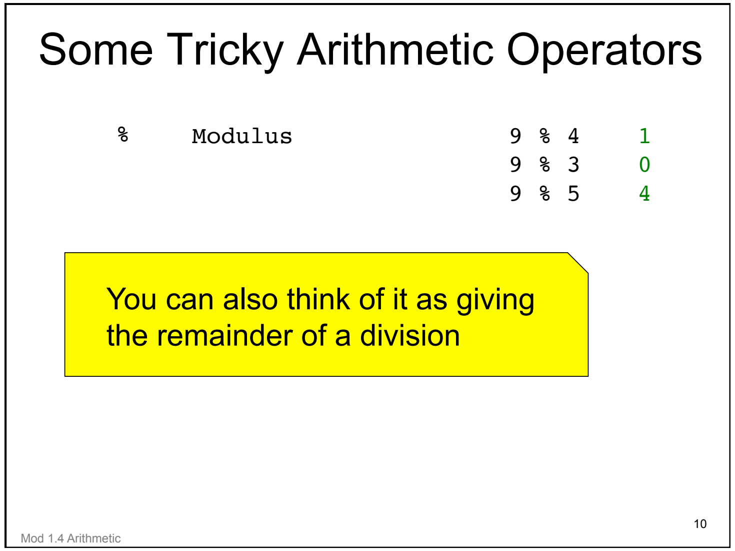


Arithmetic Operators

Notice that the result of dividing two integers may not be what you expect

3/2	1
1/2	0

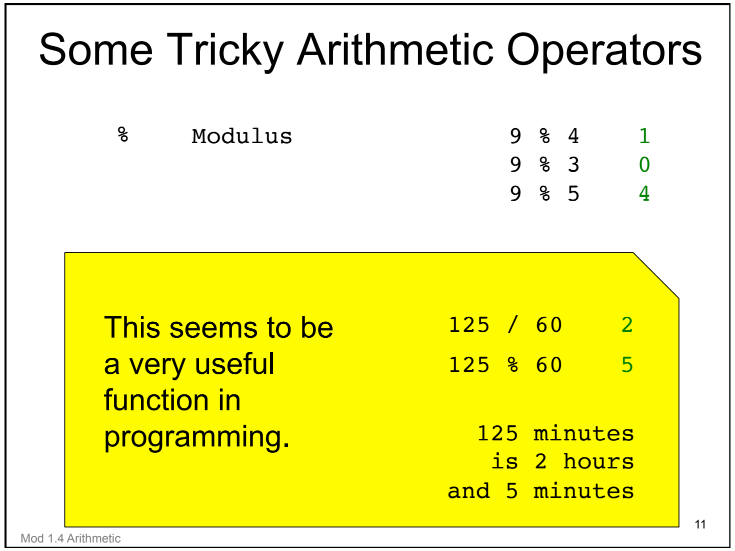
/	Division	3.0/2 3/2	1.5 1



Some Tricky Arithmetic Operators

%	Modulus	9 % 4 9 % 3 9 % 5	1 0 4

You can also think of it as giving the remainder of a division



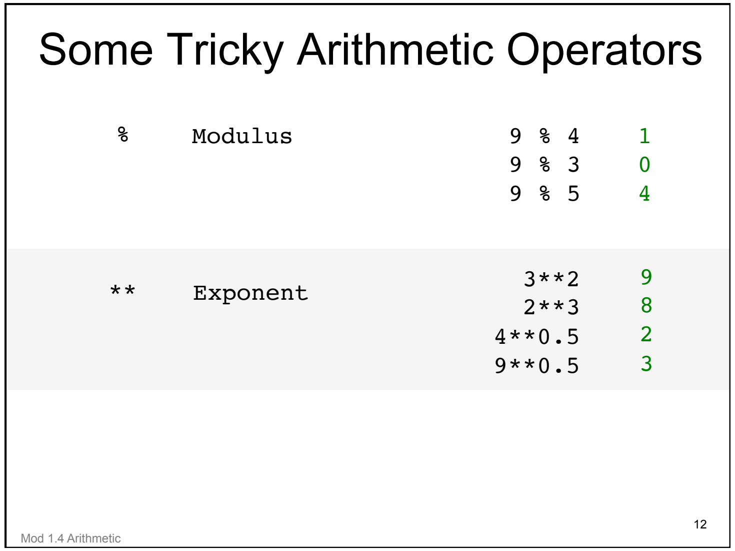
Some Tricky Arithmetic Operators

%	Modulus	9 % 4 9 % 3 9 % 5	1 0 4

This seems to be a very useful function in programming.

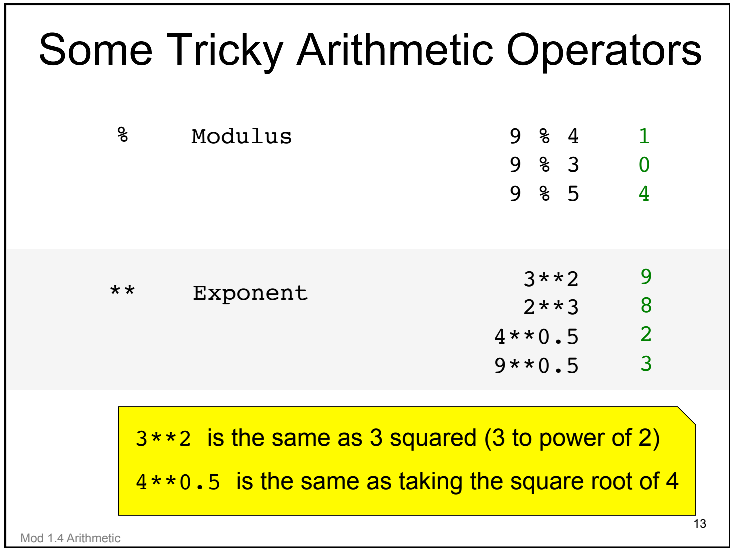
125 / 60	2
125 % 60	5

125 minutes is 2 hours and 5 minutes



Some Tricky Arithmetic Operators

%	Modulus	9 % 4 9 % 3 9 % 5	1 0 4
**	Exponent	3**2 2**3 4**0.5 9**0.5	9 8 2 3

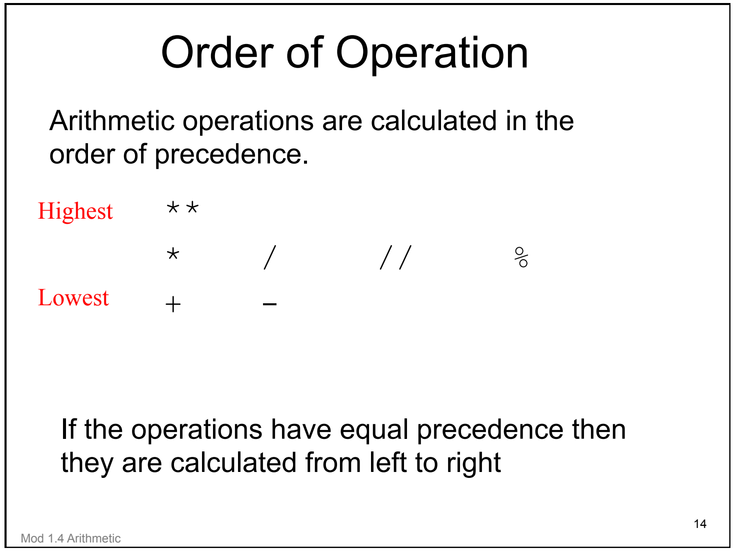


Some Tricky Arithmetic Operators

`%`	`Modulus`	`9 % 4`	1
		`9 % 3`	0
		`9 % 5`	4
`**`	`Exponent`	`3**2`	9
		`2**3`	8
		`4**0.5`	2
		`9**0.5`	3

`3**2` is the same as 3 squared (3 to power of 2)

`4**0.5` is the same as taking the square root of 4

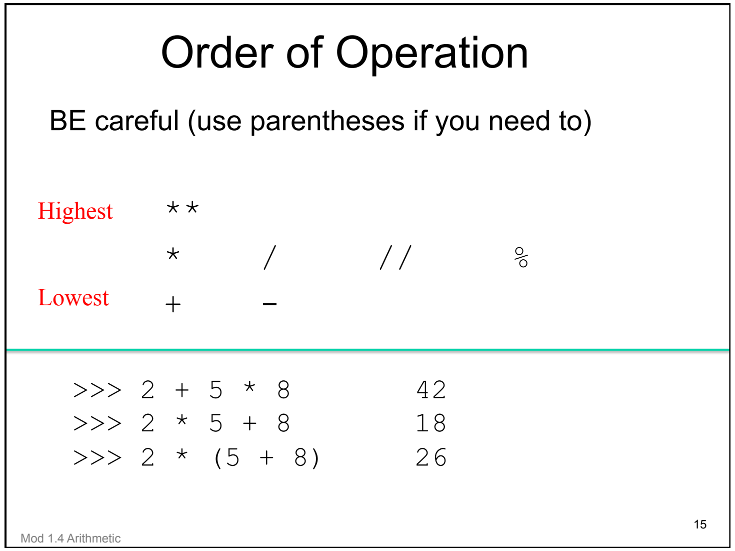


Order of Operation

Arithmetic operations are calculated in the order of precedence.

```
Highest   **
          *     /     //     %
Lowest    +     -
```

If the operations have equal precedence then they are calculated from left to right

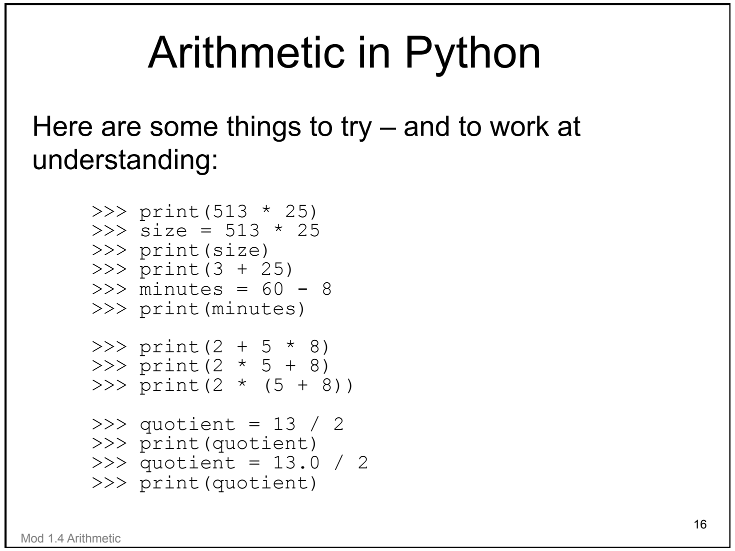


Order of Operation

BE careful (use parentheses if you need to)

```
Highest   **
          *     /     //     %
Lowest    +     -
```

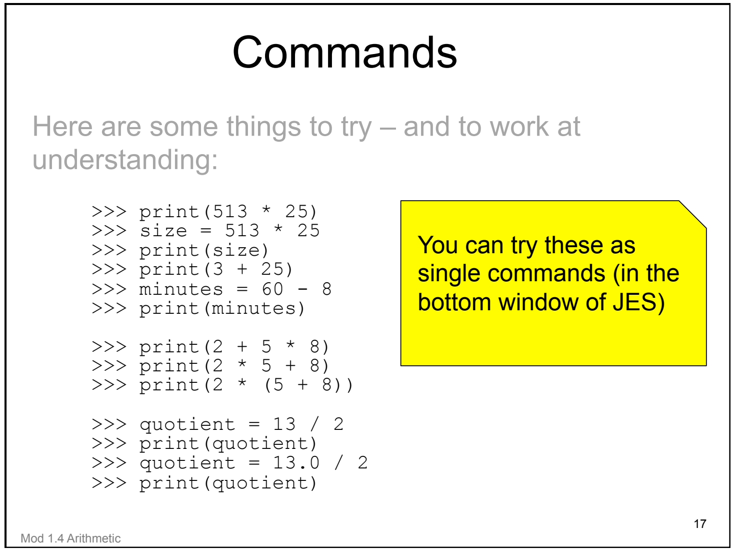
```
>>> 2 + 5 * 8         42
>>> 2 * 5 + 8         18
>>> 2 * (5 + 8)       26
```



Arithmetic in Python

Here are some things to try – and to work at understanding:

```
>>> print(513 * 25)
>>> size = 513 * 25
>>> print(size)
>>> print(3 + 25)
>>> minutes = 60 - 8
>>> print(minutes)

>>> print(2 + 5 * 8)
>>> print(2 * 5 + 8)
>>> print(2 * (5 + 8))

>>> quotient = 13 / 2
>>> print(quotient)
>>> quotient = 13.0 / 2
>>> print(quotient)
```

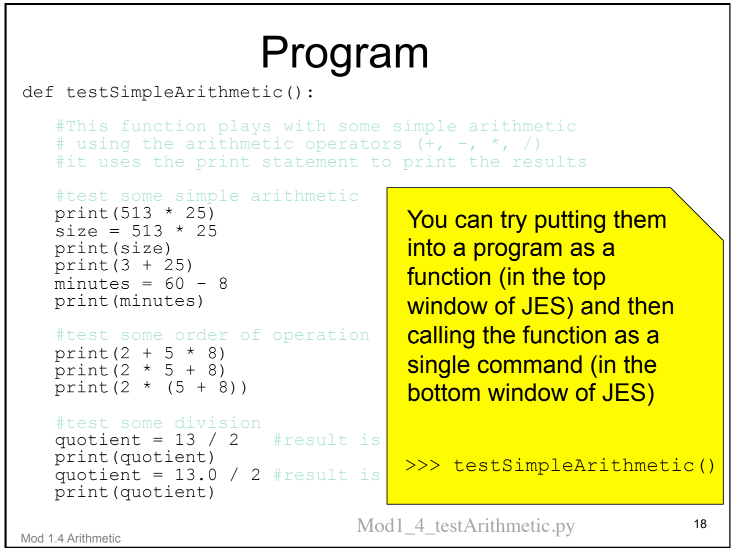


Commands

Here are some things to try – and to work at understanding:

```
>>> print(513 * 25)
>>> size = 513 * 25
>>> print(size)
>>> print(3 + 25)
>>> minutes = 60 - 8
>>> print(minutes)

>>> print(2 + 5 * 8)
>>> print(2 * 5 + 8)
>>> print(2 * (5 + 8))

>>> quotient = 13 / 2
>>> print(quotient)
>>> quotient = 13.0 / 2
>>> print(quotient)
```

You can try these as single commands (in the bottom window of JES)



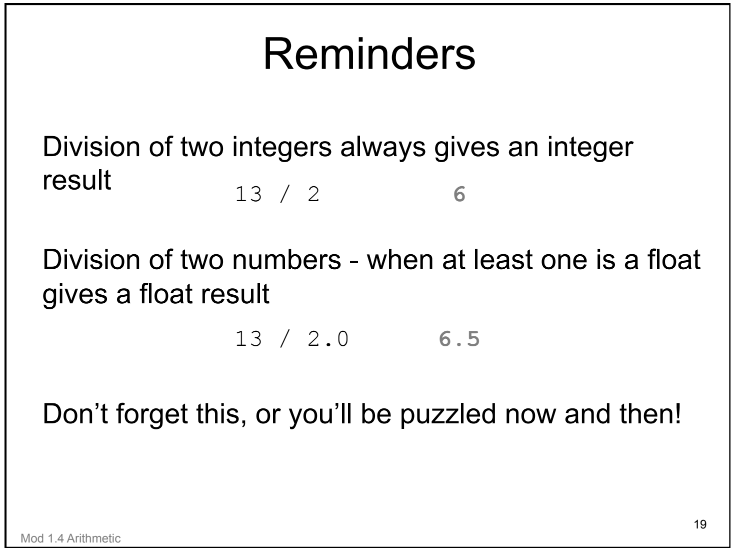
Program

```
def testSimpleArithmetic():

   #This function plays with some simple arithmetic
   # using the arithmetic operators (+, -, *, /)
   #it uses the print statement to print the results

   #test some simple arithmetic
   print(513 * 25)
   size = 513 * 25
   print(size)
   print(3 + 25)
   minutes = 60 - 8
   print(minutes)

   #test some order of operation
   print(2 + 5 * 8)
   print(2 * 5 + 8)
   print(2 * (5 + 8))

   #test some division
   quotient = 13 / 2   #result is
   print(quotient)
   quotient = 13.0 / 2 #result is
   print(quotient)
```

You can try putting them into a program as a function (in the top window of JES) and then calling the function as a single command (in the bottom window of JES)

```
>>> testSimpleArithmetic()
```

Mod1_4_testArithmetic.py



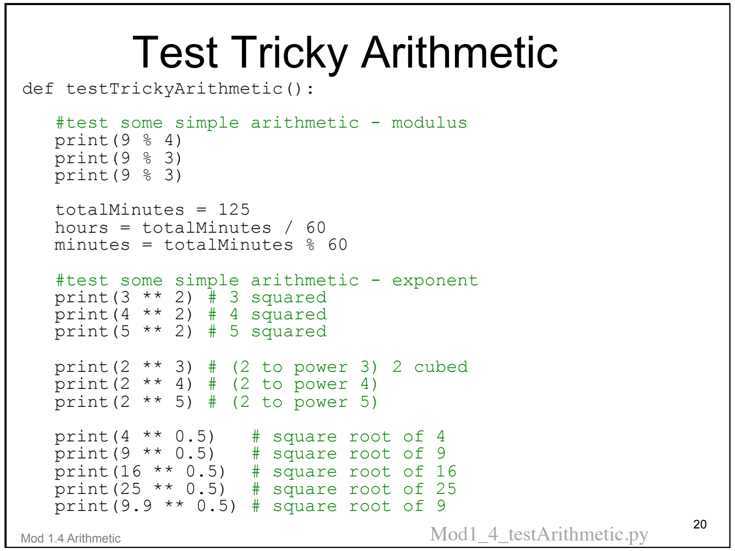
Reminders

Division of two integers always gives an integer result `13 / 2` `6`

Division of two numbers - when at least one is a float gives a float result

`13 / 2.0` `6.5`

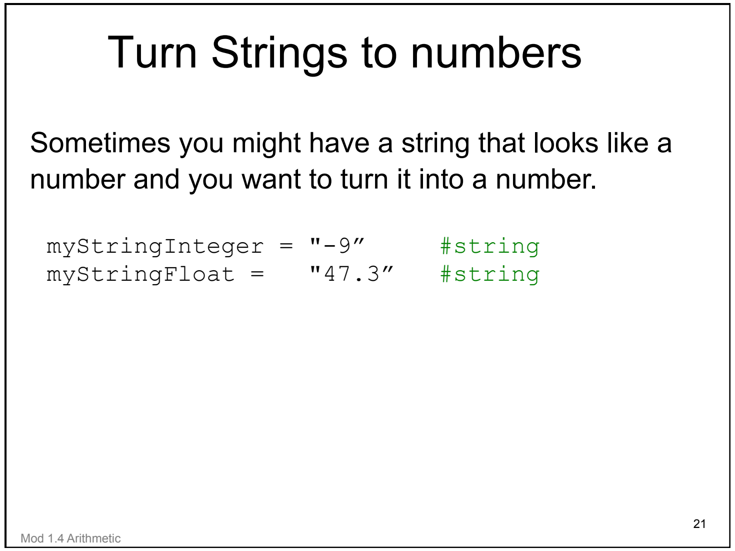
Don’t forget this, or you’ll be puzzled now and then!



Test Tricky Arithmetic

```
def testTrickyArithmetic():

   #test some simple arithmetic - modulus
   print(9 % 4)
   print(9 % 3)
   print(9 % 3)

   totalMinutes = 125
   hours = totalMinutes / 60
   minutes = totalMinutes % 60

   #test some simple arithmetic - exponent
   print(3 ** 2) # 3 squared
   print(4 ** 2) # 4 squared
   print(5 ** 2) # 5 squared

   print(2 ** 3) # (2 to power 3) 2 cubed
   print(2 ** 4) # (2 to power 4)
   print(2 ** 5) # (2 to power 5)

   print(4 ** 0.5)    # square root of 4
   print(9 ** 0.5)    # square root of 9
   print(16 ** 0.5)   # square root of 16
   print(25 ** 0.5)   # square root of 25
   print(9.9 ** 0.5)  # square root of 9
```

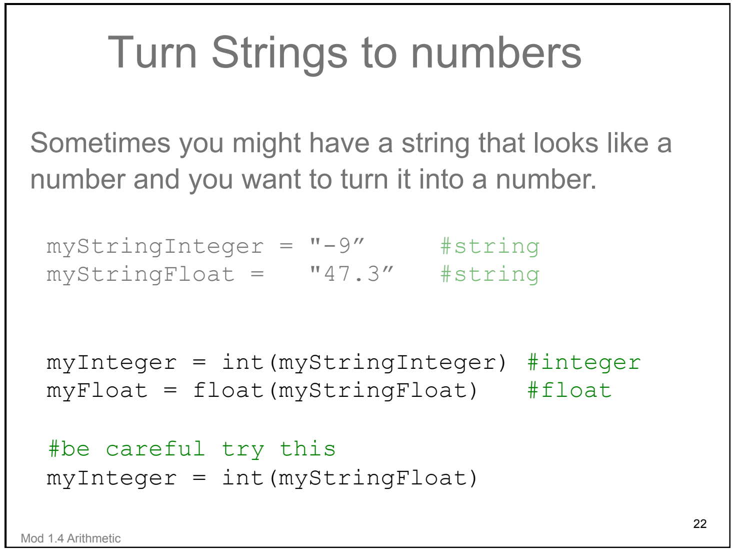
Mod1_4_testArithmetic.py



Turn Strings to numbers

Sometimes you might have a string that looks like a number and you want to turn it into a number.

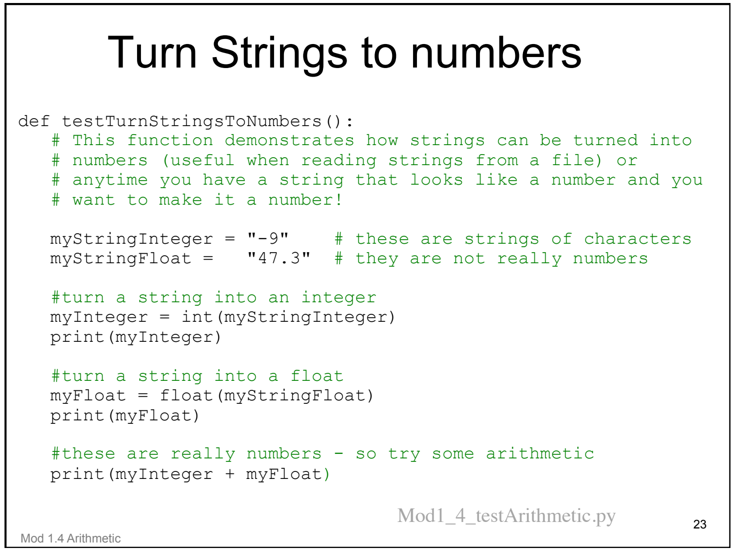
```
myStringInteger = "-9”      #string
myStringFloat =   "47.3”    #string
```



Turn Strings to numbers

Sometimes you might have a string that looks like a number and you want to turn it into a number.

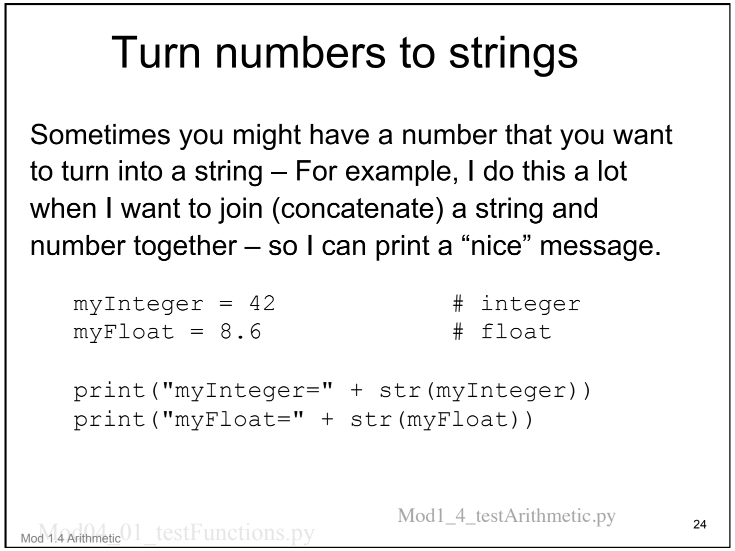
```
myStringInteger = "-9”      #string
myStringFloat =   "47.3”    #string


myInteger = int(myStringInteger)  #integer
myFloat = float(myStringFloat)    #float

#be careful try this
myInteger = int(myStringFloat)
```



Turn Strings to numbers

```
def testTurnStringsToNumbers():
   # This function demonstrates how strings can be turned into
   # numbers (useful when reading strings from a file) or
   # anytime you have a string that looks like a number and you
   # want to make it a number!

   myStringInteger = "-9"    # these are strings of characters
   myStringFloat =   "47.3"  # they are not really numbers

   #turn a string into an integer
   myInteger = int(myStringInteger)
   print(myInteger)

   #turn a string into a float
   myFloat = float(myStringFloat)
   print(myFloat)

   #these are really numbers - so try some arithmetic
   print(myInteger + myFloat)
```

Mod1_4_testArithmetic.py



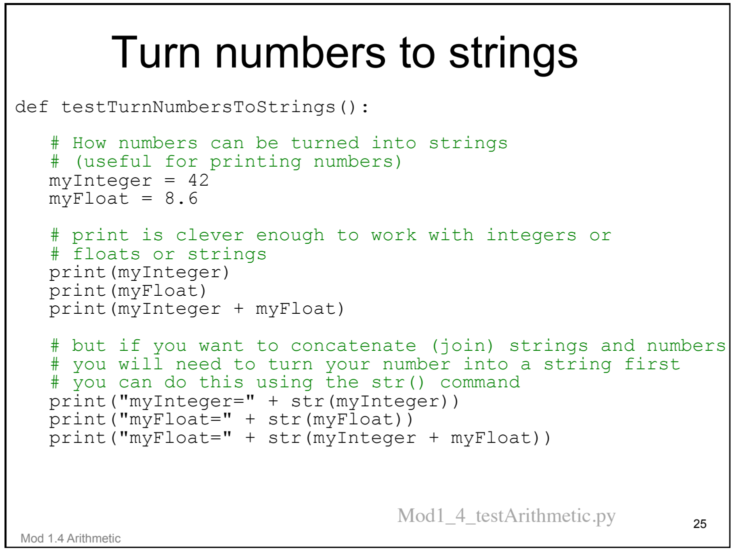
Turn numbers to strings

Sometimes you might have a number that you want to turn into a string – For example, I do this a lot when I want to join (concatenate) a string and number together – so I can print a “nice” message.

```
myInteger = 42              # integer
myFloat = 8.6               # float

print("myInteger=" + str(myInteger))
print("myFloat=" + str(myFloat))
```

Mod1_4_testArithmetic.py

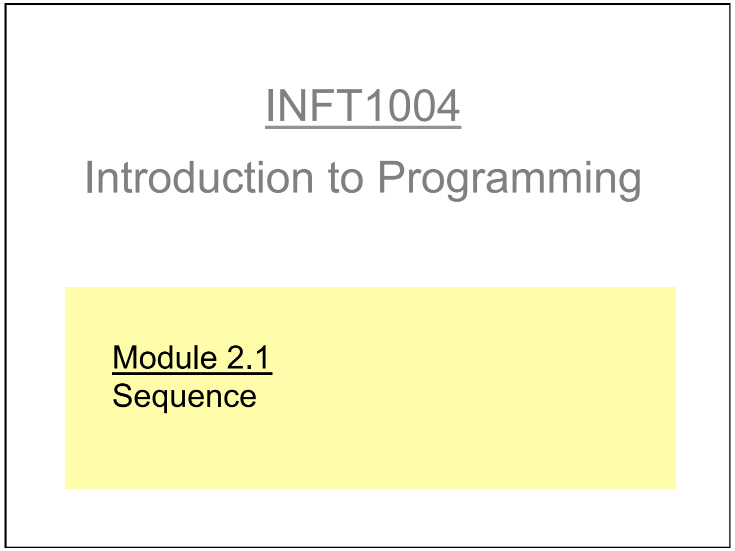
Mod04_01_testFunctions.py



Turn numbers to strings

```
def testTurnNumbersToStrings():

   # How numbers can be turned into strings
   # (useful for printing numbers)
   myInteger = 42
   myFloat = 8.6

   # print is clever enough to work with integers or
   # floats or strings
   print(myInteger)
   print(myFloat)
   print(myInteger + myFloat)

   # but if you want to concatenate (join) strings and numbers
   # you will need to turn your number into a string first
   # you can do this using the str() command
   print("myInteger=" + str(myInteger))
   print("myFloat=" + str(myFloat))
   print("myFloat=" + str(myInteger + myFloat))
```

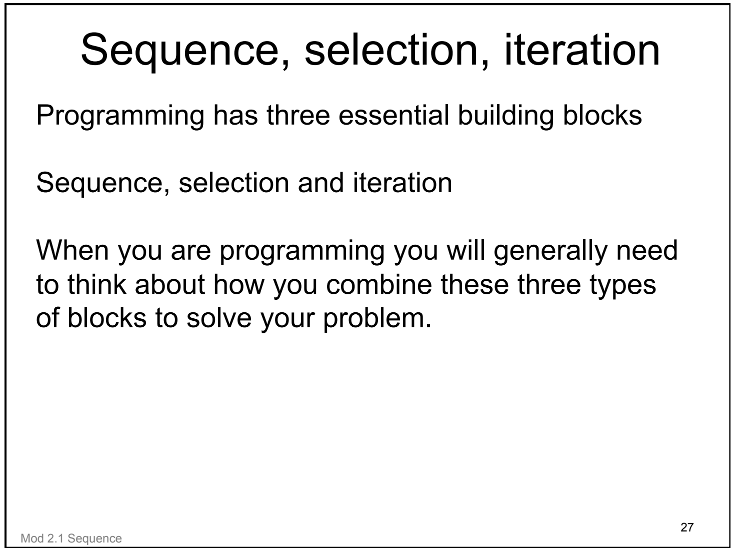
Mod1_4_testArithmetic.py



INFT1004

Introduction to Programming

Module 2.1
Sequence

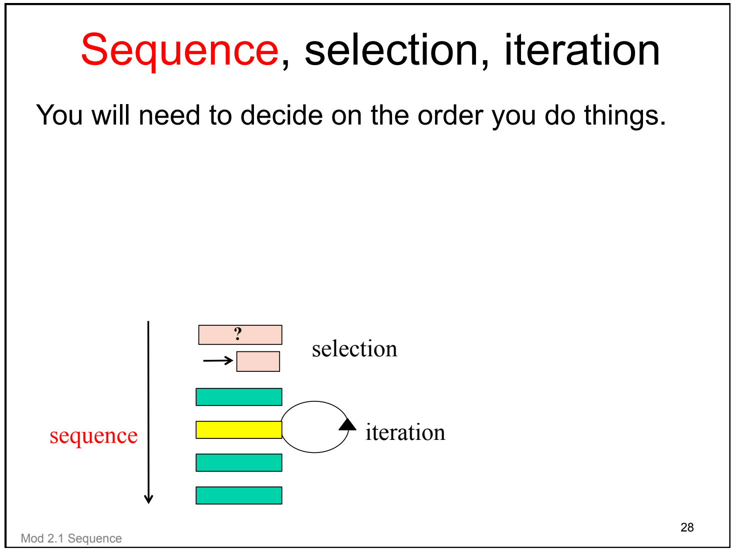


Sequence, selection, iteration

Programming has three essential building blocks

Sequence, selection and iteration

When you are programming you will generally need to think about how you combine these three types of blocks to solve your problem.



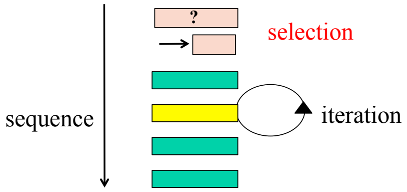
Sequence, selection, iteration

You will need to decide on the order you do things.

Sequence, selection, iteration

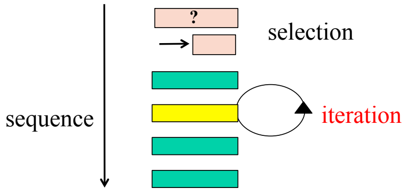
You may need to do different things depending on some kind of condition

You might skip code, do extra things, or perhaps different things depending on different conditions)



Sequence, selection, iteration

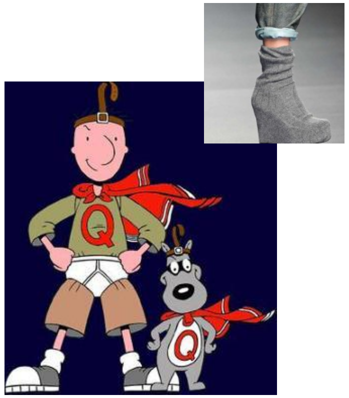
You will often want to repeat things or process long lists of things all the same. This is sometimes called looping.



Today– sequence

Something we mentioned in the first week is the important concept of sequence

Programming is like dressing – the order you do things can be important!

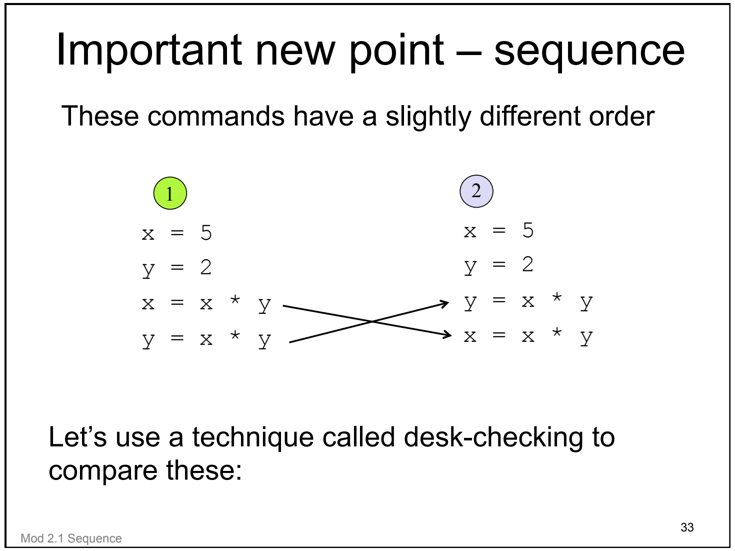


Important new point – sequence

If the body of a function or a *for* statement has, say, 10 statements, they will be executed in the order they’re written

Order can be very important!

The same statements in a different order can do different things



Important new point – sequence

These commands have a slightly different order

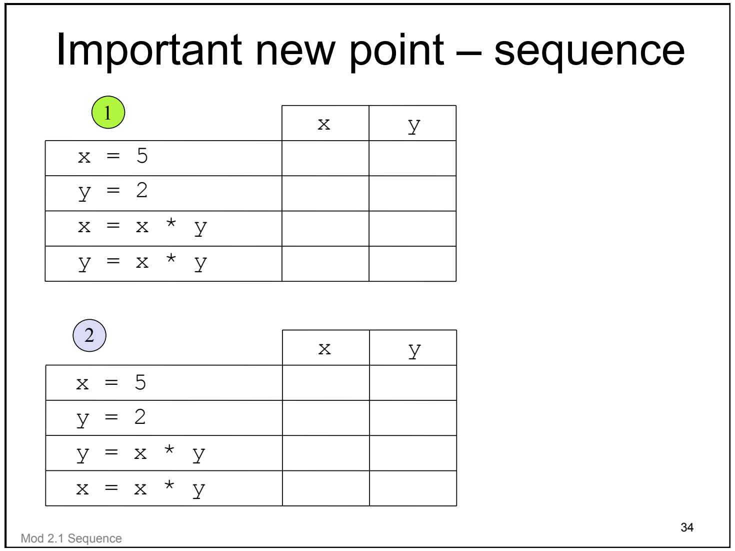
1

```
x = 5
y = 2
x = x * y
y = x * y
```

2

```
x = 5
y = 2
y = x * y
x = x * y
```

Let’s use a technique called desk-checking to compare these:



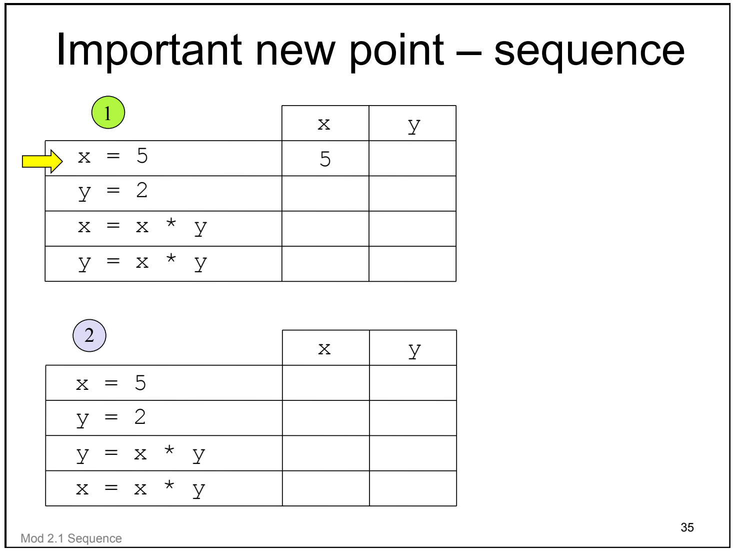
Important new point – sequence

1

	x	y
x = 5		
y = 2		
x = x * y		
y = x * y		

2

	x	y
x = 5		
y = 2		
y = x * y		
x = x * y		



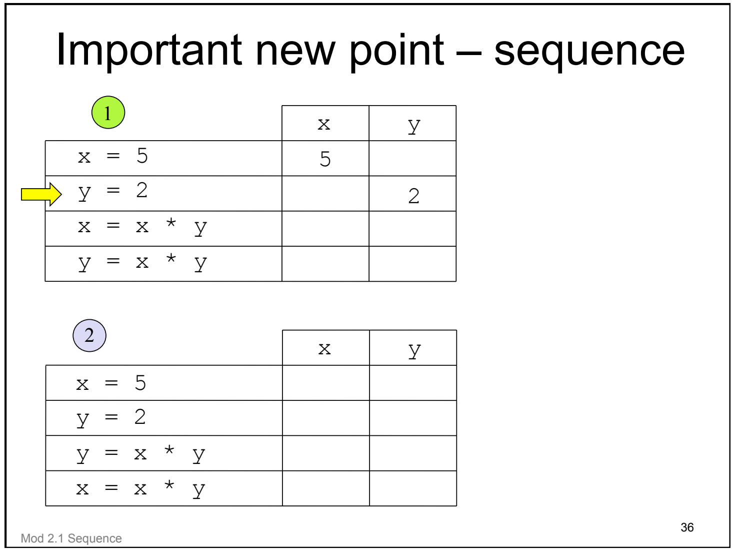
Important new point – sequence

1

	x	y
⇨ x = 5	5	
y = 2		
x = x * y		
y = x * y		

2

	x	y
x = 5		
y = 2		
y = x * y		
x = x * y		



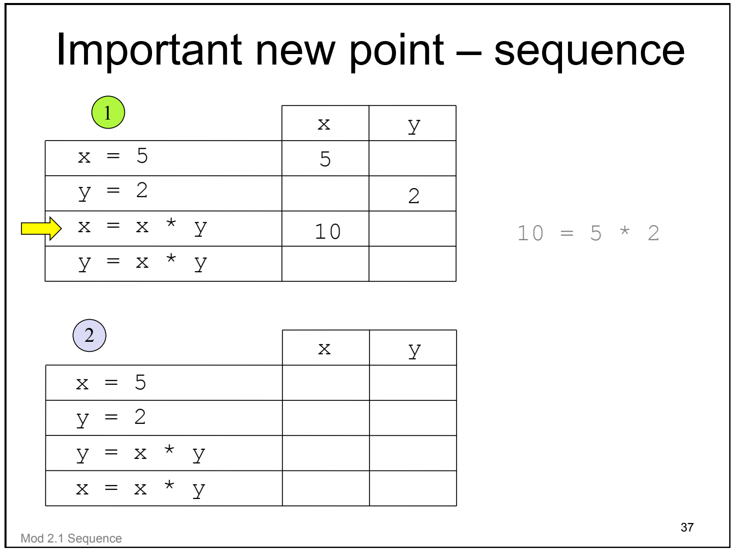
Important new point – sequence

1

	x	y
x = 5	5	
⇨ y = 2		2
x = x * y		
y = x * y		

2

	x	y
x = 5		
y = 2		
y = x * y		
x = x * y		



Important new point – sequence

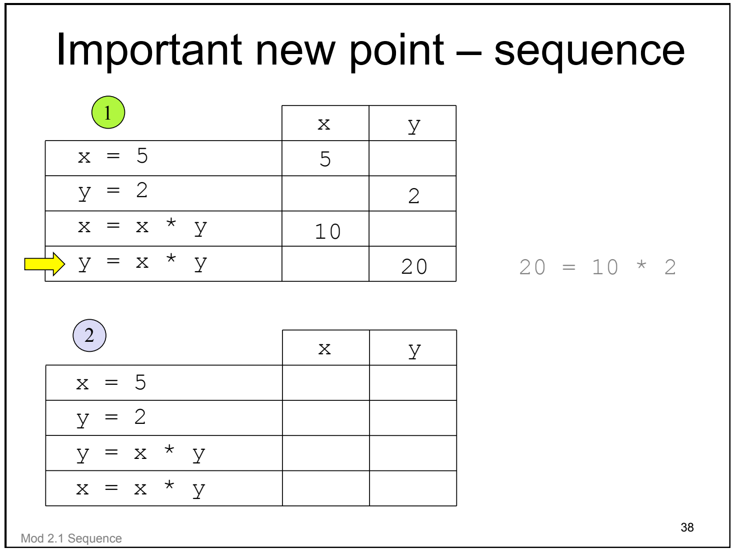
1

	x	y
x = 5	5	
y = 2		2
⇨ x = x * y	10	
y = x * y		

10 = 5 * 2

2

	x	y
x = 5		
y = 2		
y = x * y		
x = x * y		



Important new point – sequence

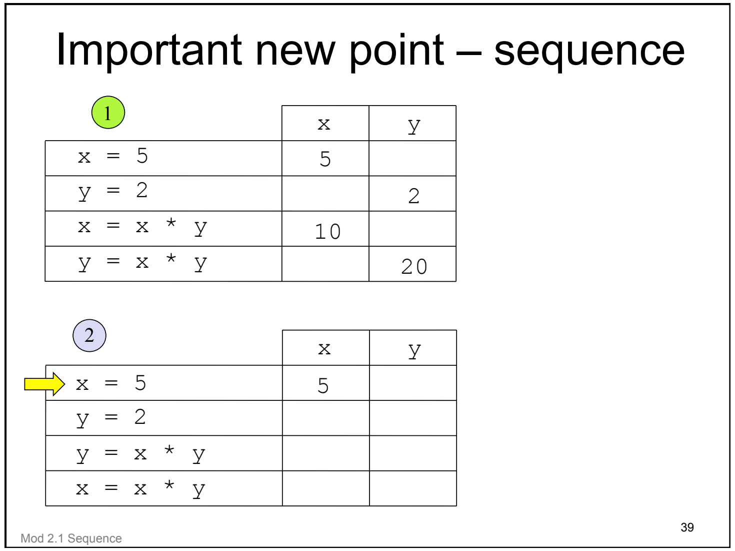
1

	x	y
x = 5	5	
y = 2		2
x = x * y	10	
⇨ y = x * y		20

20 = 10 * 2

2

	x	y
x = 5		
y = 2		
y = x * y		
x = x * y		



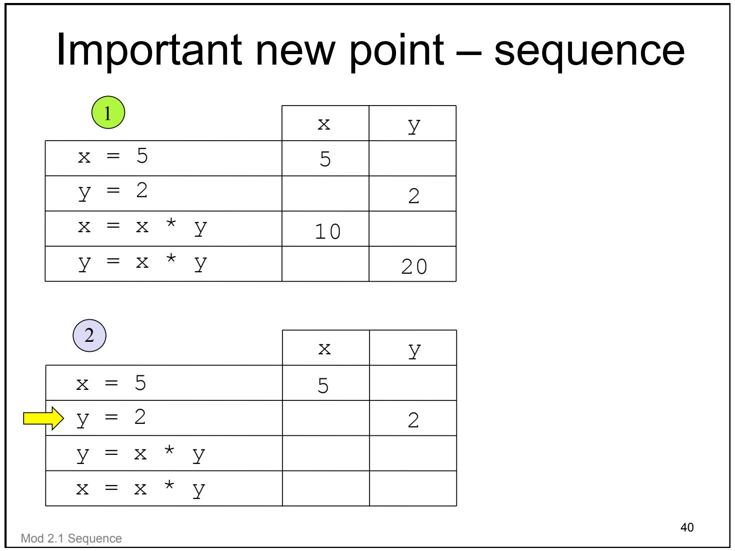
Important new point – sequence

1

	x	y
x = 5	5	
y = 2		2
x = x * y	10	
y = x * y		20

2

	x	y
⇨ x = 5	5	
y = 2		
y = x * y		
x = x * y		



Important new point – sequence

1

	x	y
x = 5	5	
y = 2		2
x = x * y	10	
y = x * y		20

2

	x	y
x = 5	5	
⇨ y = 2		2
y = x * y		
x = x * y		

Important new point – sequence

1

	x	y
x = 5	5	
y = 2		2
x = x * y	10	
y = x * y		20

2

	x	y
x = 5	5	
y = 2		2
⇨ y = x * y		10
x = x * y		

10 = 5 * 2

Important new point – sequence

1

	x	y
x = 5	5	
y = 2		2
x = x * y	10	
y = x * y		20

2

	x	y
x = 5	5	
y = 2		2
y = x * y		10
⇨ x = x * y	50	

50 = 5 * 10

Important new point – sequence

1

	x	y
x = 5	5	
y = 2		2
x = x * y	10	
y = x * y		20

x is 10
y is 20

The order (sequence of instructions) is important

Just swapping two statements here creates very different answers

2

	x	y
x = 5	5	
y = 2		2
y = x * y		10
x = x * y	50	

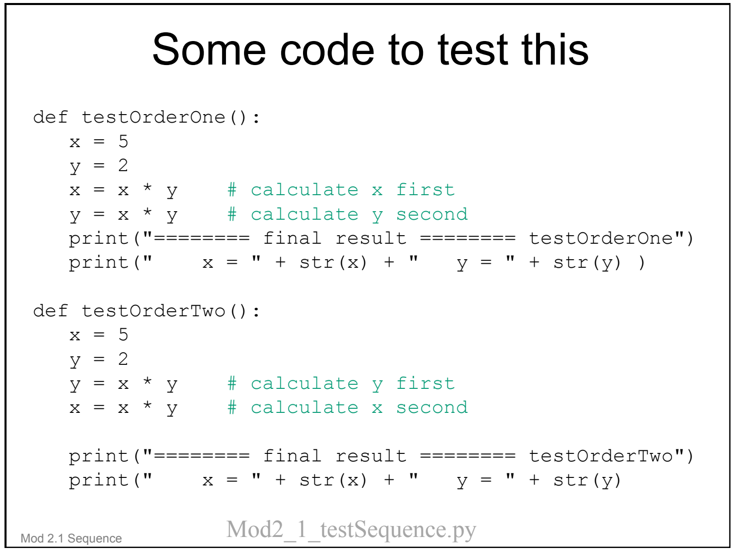
x is 50
y is 10

Learn to Desk-check

	x	y
x = 5	5	
y = 2		2
x = x * y	10	
y = x * y		20

Desk Checking is an excellent way to debug your programs. (It' s easy and low tech)

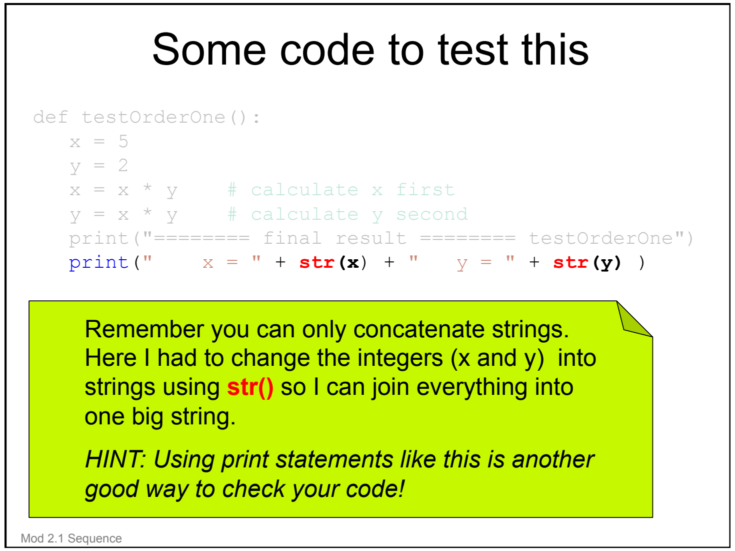
It can be a very helpful way to work out answers in an Exam!



Some code to test this

```
def testOrderOne():
   x = 5
   y = 2
   x = x * y    # calculate x first
   y = x * y    # calculate y second
   print("======== final result ======== testOrderOne")
   print("    x = " + str(x) + "    y = " + str(y) )

def testOrderTwo():
   x = 5
   y = 2
   y = x * y    # calculate y first
   x = x * y    # calculate x second

   print("======== final result ======== testOrderTwo")
   print("    x = " + str(x) + "    y = " + str(y)
```

Mod2_1_testSequence.py

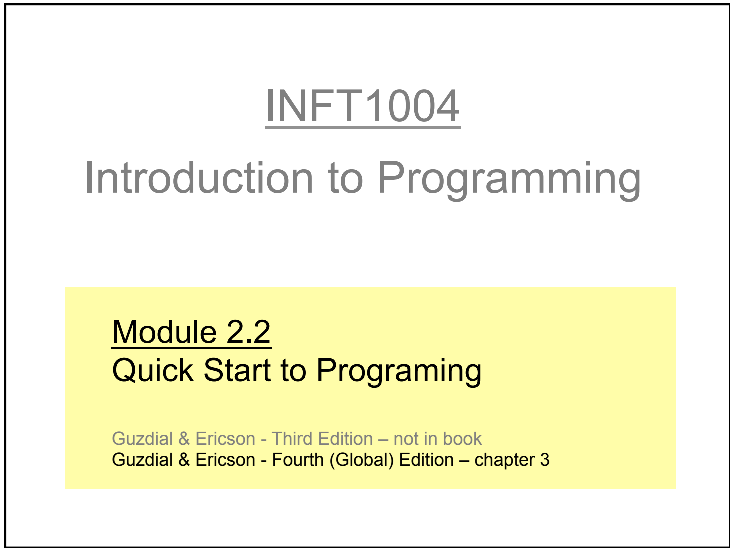


Some code to test this

```
def testOrderOne():
   x = 5
   y = 2
   x = x * y    # calculate x first
   y = x * y    # calculate y second
   print("======== final result ======== testOrderOne")
   print("    x = " + str(x) + "    y = " + str(y) )
```

Remember you can only concatenate strings. Here I had to change the integers (x and y) into strings using str() so I can join everything into one big string.

HINT: Using print statements like this is another good way to check your code!

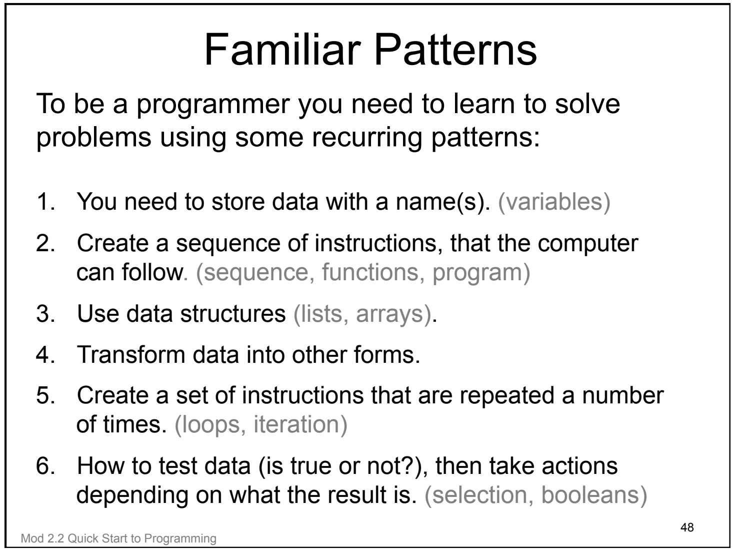


INFT1004

Introduction to Programming

Module 2.2
Quick Start to Programing

Guzdial & Ericson - Third Edition – not in book
Guzdial & Ericson - Fourth (Global) Edition – chapter 3



Familiar Patterns

To be a programmer you need to learn to solve problems using some recurring patterns:

1. You need to store data with a name(s). (variables)
2. Create a sequence of instructions, that the computer can follow. (sequence, functions, program)
3. Use data structures (lists, arrays).
4. Transform data into other forms.
5. Create a set of instructions that are repeated a number of times. (loops, iteration)
6. How to test data (is true or not?), then take actions depending on what the result is. (selection, booleans)

Familiar Patterns - Strings

This module gives you a quick start to using these tools – it uses strings as an example.

1. You need to store data with a name(s). (variables)
2. Create a sequence of instructions, that the computer can follow. (sequence, functions, program)
3. Use data structures (lists, arrays).
4. Transform data into other forms.
5. Create a set of instructions that are repeated a number of times. (loops, iteration)
6. How to test data (is true or not?), then take actions depending on what the result is. (selection, booleans)

One Type of Data - Strings

Strings are defined with quote marks.

Python supports three kinds of quotes:

```
>>> print 'this is a string'
this is a string
>>> print "this is a string"
this is a string
>>> print """this is a string"""
this is a string
```

You need to use right one if you need to embed quote marks eg.

```
>>> aSingleQuote = "   '    "
>>> print aSingleQuote
   '
```

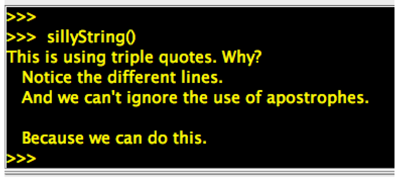
sillyString()

```
def sillyString():
   print """This is using triple quotes. Why?
   Notice the different lines.
   And we can't ignore the use of apostrophes.

   Because we can do this."""
```

This is a function that has been defined. Most of the time you will write your own functions to do something useful. Try to give them a meaningful name.

sillyString()

```
def sillyString():
   print """This is using triple quotes. Why?
   Notice the different lines.
   And we can't ignore the use of apostrophes.

   Because we can do this."""
```



```
>>>
>>> sillyString()
This is using triple quotes. Why?
  Notice the different lines.
  And we can't ignore the use of apostrophes.

  Because we can do this.
>>>
```

You can then use your function as a command.

Some other data types

A ***String*** (str) stores a sequence of characters in memory.

Integer (int) and ***Floating Point*** (float) are types used to store numbers in memory.

Floating Point types can code decimal places eg. 4.56

Converting data types

You can add numbers together (int, float)
You can join strings together

You can’t add strings and numbers (int, float) together
You can’t join strings with numbers (int, float)

```
>>> “4” + 3
```

If you want to do this you (and you often do) then you need to convert first.

Converting data types

To convert a number (int, float) to a string use the function str()

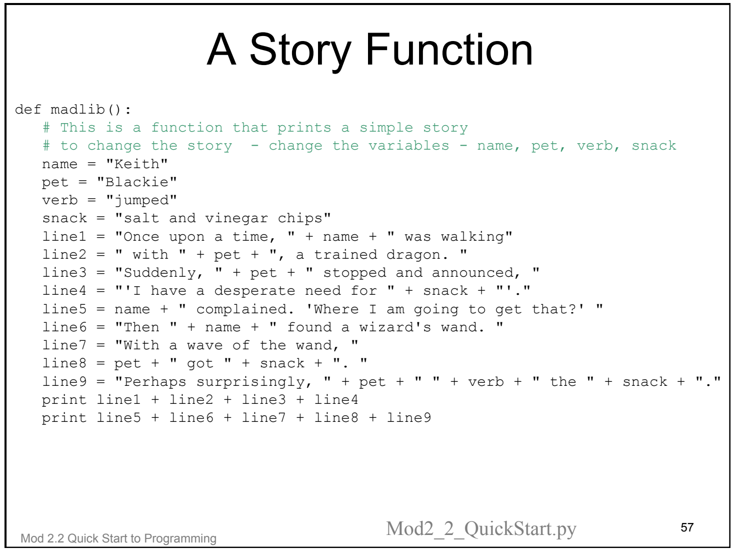
```
>>> “4” + str(3)

    “43”

>>> “4” + str(3.2)

    “43.2”
```

Converting data types

To convert a string to a number use either int() or float()

```
>>> int(“4”) + 3

   7

>>> float(“4.2”) + 3

    7.2
```

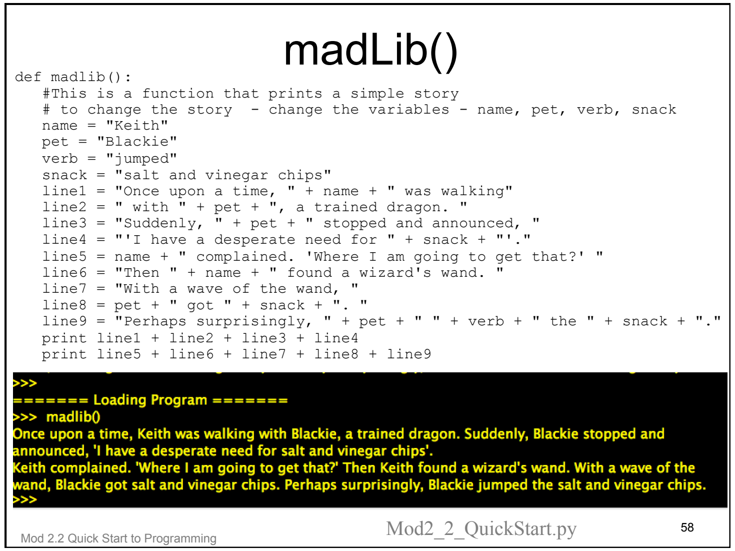
You need to make sure the string you want to convert can actually be turned into a number
e.g. int(“abc”) will not work



A Story Function

```
def madlib():
   # This is a function that prints a simple story
   # to change the story  - change the variables - name, pet, verb, snack
   name = "Keith"
   pet = "Blackie"
   verb = "jumped"
   snack = "salt and vinegar chips"
   line1 = "Once upon a time, " + name + " was walking"
   line2 = " with " + pet + ", a trained dragon. "
   line3 = "Suddenly, " + pet + " stopped and announced, "
   line4 = "'I have a desperate need for " + snack + "'."
   line5 = name + " complained. 'Where I am going to get that?' "
   line6 = "Then " + name + " found a wizard's wand. "
   line7 = "With a wave of the wand, "
   line8 = pet + " got " + snack + ". "
   line9 = "Perhaps surprisingly, " + pet + " " + verb + " the " + snack + "."
   print line1 + line2 + line3 + line4
   print line5 + line6 + line7 + line8 + line9
```

Mod2_2_QuickStart.py

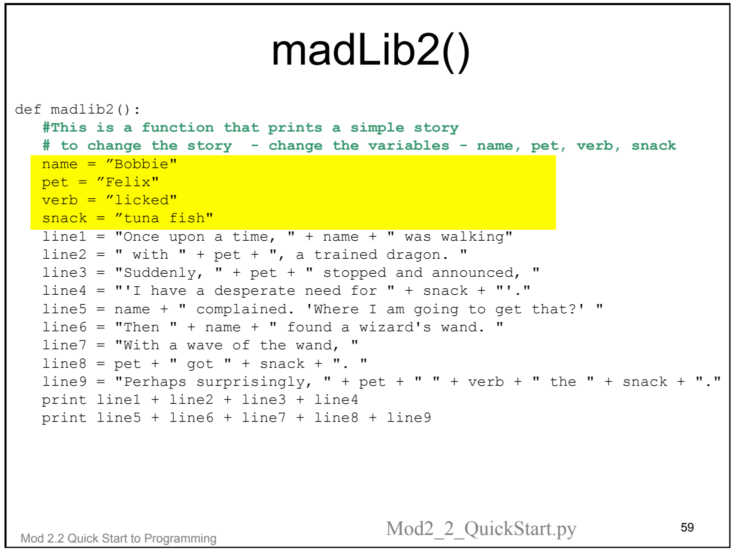


madLib()

```
def madlib():
   #This is a function that prints a simple story
   # to change the story  - change the variables - name, pet, verb, snack
   name = "Keith"
   pet = "Blackie"
   verb = "jumped"
   snack = "salt and vinegar chips"
   line1 = "Once upon a time, " + name + " was walking"
   line2 = " with " + pet + ", a trained dragon. "
   line3 = "Suddenly, " + pet + " stopped and announced, "
   line4 = "'I have a desperate need for " + snack + "'."
   line5 = name + " complained. 'Where I am going to get that?' "
   line6 = "Then " + name + " found a wizard's wand. "
   line7 = "With a wave of the wand, "
   line8 = pet + " got " + snack + ". "
   line9 = "Perhaps surprisingly, " + pet + " " + verb + " the " + snack + "."
   print line1 + line2 + line3 + line4
   print line5 + line6 + line7 + line8 + line9
```

```
>>>
======= Loading Program =======
>>>  madlib()
Once upon a time, Keith was walking with Blackie, a trained dragon. Suddenly, Blackie stopped and
announced, 'I have a desperate need for salt and vinegar chips'.
Keith complained. 'Where I am going to get that?' Then Keith found a wizard's wand. With a wave of the
wand, Blackie got salt and vinegar chips. Perhaps surprisingly, Blackie jumped the salt and vinegar chips.
>>>
```

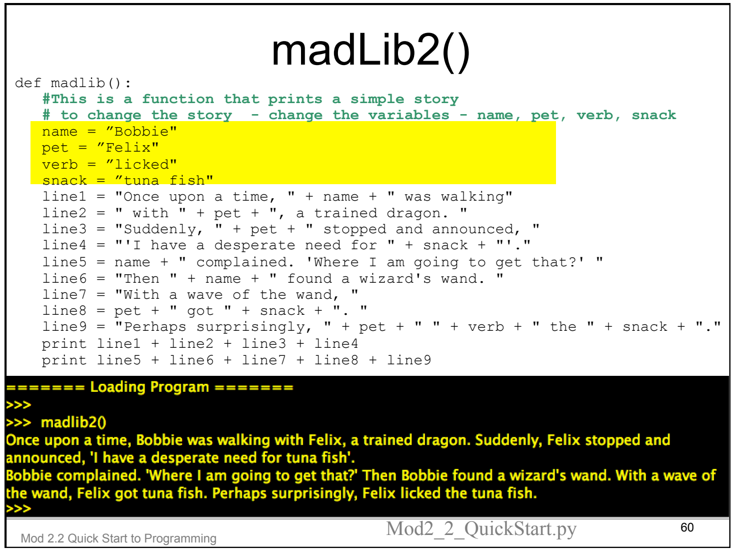
Mod2_2_QuickStart.py



madLib2()

```
def madlib2():
   #This is a function that prints a simple story
   # to change the story  - change the variables - name, pet, verb, snack
   name = ”Bobbie"
   pet = ”Felix"
   verb = ”licked"
   snack = ”tuna fish"
   line1 = "Once upon a time, " + name + " was walking"
   line2 = " with " + pet + ", a trained dragon. "
   line3 = "Suddenly, " + pet + " stopped and announced, "
   line4 = "'I have a desperate need for " + snack + "'."
   line5 = name + " complained. 'Where I am going to get that?' "
   line6 = "Then " + name + " found a wizard's wand. "
   line7 = "With a wave of the wand, "
   line8 = pet + " got " + snack + ". "
   line9 = "Perhaps surprisingly, " + pet + " " + verb + " the " + snack + "."
   print line1 + line2 + line3 + line4
   print line5 + line6 + line7 + line8 + line9
```

Mod2_2_QuickStart.py



madLib2()

```
def madlib():
   #This is a function that prints a simple story
   # to change the story  - change the variables - name, pet, verb, snack
   name = ”Bobbie"
   pet = ”Felix"
   verb = ”licked"
   snack = ”tuna fish"
   line1 = "Once upon a time, " + name + " was walking"
   line2 = " with " + pet + ", a trained dragon. "
   line3 = "Suddenly, " + pet + " stopped and announced, "
   line4 = "'I have a desperate need for " + snack + "'."
   line5 = name + " complained. 'Where I am going to get that?' "
   line6 = "Then " + name + " found a wizard's wand. "
   line7 = "With a wave of the wand, "
   line8 = pet + " got " + snack + ". "
   line9 = "Perhaps surprisingly, " + pet + " " + verb + " the " + snack + "."
   print line1 + line2 + line3 + line4
   print line5 + line6 + line7 + line8 + line9
```

```
======= Loading Program =======
>>>
>>>  madlib2()
Once upon a time, Bobbie was walking with Felix, a trained dragon. Suddenly, Felix stopped and
announced, 'I have a desperate need for tuna fish'.
Bobbie complained. 'Where I am going to get that?' Then Bobbie found a wizard's wand. With a wave of
the wand, Felix got tuna fish. Perhaps surprisingly, Felix licked the tuna fish.
>>>
```

Mod2_2_QuickStart.py

madLib2() - problems

```
def madlib3():
    #This is a function that prints a simple story
    # to change the story  - change the variables - name, pet, verb, snack
    name = ”Bobbie"
    pet = ”Felix"
    verb = ”licked"
    snack = ”tuna fish"
    .
    .
    .
```

The problem with changing these variables is that you have to change the code each time.

There is no reuse (even though most of the code in madlib and madlib2 is the same)

Solution – change the variables to parameters

Mod2_2_QuickStart.py

madLib3() - parameters

```
def madlib3(name, pet, verb, snack):
    #This is a function that prints a simple story
    # to change the story  - use the parameters - name, pet, verb, snack
    # name = ”Bobbie"
    # pet = ”Felix"
    # verb = ”licked"
    # snack = ”tuna fish"
    line1 = "Once upon a time, " + name + " was walking"
    line2 = " with " + pet + ", a trained dragon. "
    line3 = "Suddenly, " + pet + " stopped and announced, "
    line4 = "'I have a desperate need for " + snack + "'."
    line5 = name + " complained. 'Where I am going to get that?' "
    line6 = "Then " + name + " found a wizard's wand. "
    line7 = "With a wave of the wand, "
    line8 = pet + " got " + snack + ". "
    line9 = "Perhaps surprisingly, " + pet + " " + verb + " the " + snack + "."
    print line1 + line2 + line3 + line4
    print line5 + line6 + line7 + line8 + line9
```

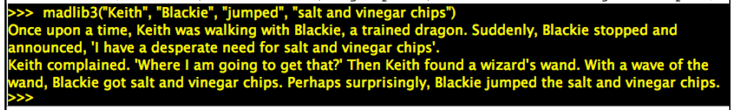
Mod2_2_QuickStart.py

calling madLib3()

```
def madlib3(name, pet, verb, snack):
```

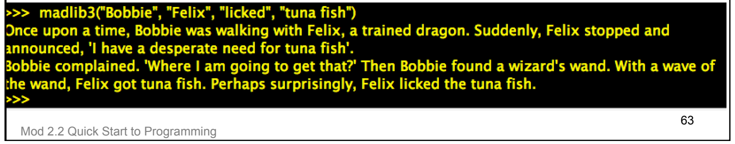
You need to provide the right arguments when calling the function

```
          name       pet       verb       snack
>>>madlib3("Keith", "Blackie", "jumped", "salt and vinegar chips")
```



```
>>>  madlib3("Keith", "Blackie", "jumped", "salt and vinegar chips")
Once upon a time, Keith was walking with Blackie, a trained dragon. Suddenly, Blackie stopped and
announced, 'I have a desperate need for salt and vinegar chips'.
Keith complained. 'Where I am going to get that?' Then Keith found a wizard's wand. With a wave of the
wand, Blackie got salt and vinegar chips. Perhaps surprisingly, Blackie jumped the salt and vinegar chips.
>>>
```

```
            name       pet       verb       snack
  >>>madlib3("Bobbie", "Felix", "licked", "tuna fish")
```

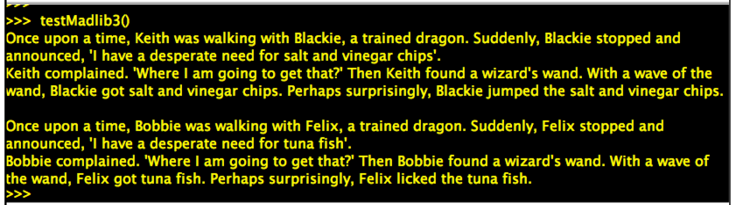


```
>>>  madlib3("Bobbie", "Felix", "licked", "tuna fish")
Once upon a time, Bobbie was walking with Felix, a trained dragon. Suddenly, Felix stopped and
announced, 'I have a desperate need for tuna fish'.
Bobbie complained. 'Where I am going to get that?' Then Bobbie found a wizard's wand. With a wave of
the wand, Felix got tuna fish. Perhaps surprisingly, Felix licked the tuna fish.
>>>
```

testing madLib3()

When I’m writing a function and want to test it I usually just write another test function (just saves time)

```
def testMadlib3():

   madlib3("Keith", "Blackie", "jumped", "salt and vinegar chips")
   print " "
   madlib3("Bobbie", "Felix", "licked", "tuna fish")
```



```
>>>
>>>  testMadlib3()
Once upon a time, Keith was walking with Blackie, a trained dragon. Suddenly, Blackie stopped and
announced, 'I have a desperate need for salt and vinegar chips'.
Keith complained. 'Where I am going to get that?' Then Keith found a wizard's wand. With a wave of the
wand, Blackie got salt and vinegar chips. Perhaps surprisingly, Blackie jumped the salt and vinegar chips.

Once upon a time, Bobbie was walking with Felix, a trained dragon. Suddenly, Felix stopped and
announced, 'I have a desperate need for tuna fish'.
Bobbie complained. 'Where I am going to get that?' Then Bobbie found a wizard's wand. With a wave of
the wand, Felix got tuna fish. Perhaps surprisingly, Felix licked the tuna fish.
>>>
```

Mod2_2_QuickStart.py

testing madLib3()

Note that this is another way of doing it – use some variables and pass the variables as arguments – this works just as well (for some people it might be clearer)

```
def anotherTestMadlib3():
   myName = "Keith"
   myPet = "Blackie"
   myVerb = "jumped"
   mySnack = "salt and vinegar chips"

   madlib3(myName, myPet, myVerb, mySnack)
   print " " #just prints an empty space

   yName = "Bobbie"
   myPet = "Felix"
   myVerb = "licked"
   mySnack = "tuna fish"
   madlib3(myName, myPet, myVerb, mySnack)
```

Mod2_2_QuickStart.py

A common pattern

The important thing is to recognise this pattern.

When you are reusing most of the same code – with a few things that change.

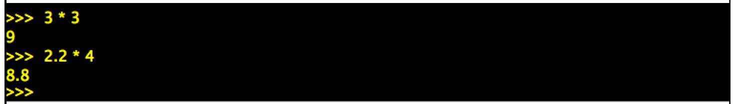
Write a function and reuse most of the code. The parts that change become parameters.

The best thing about this is you will need to write less code (and once the function is tested you can be confident it will always work)

What about multiplication

This works as expected

```
>>> 3 * 3
9

>> 2.2 * 4
8.8
```



```
>>> 3 * 3
9
>>> 2.2 * 4
8.8
>>>
```

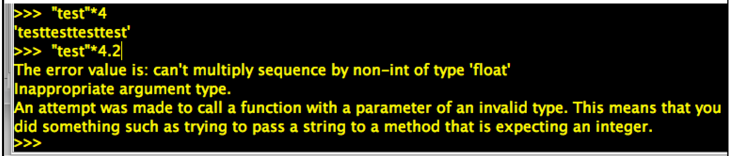
What about multiplication

This also works in python

```
>>> “test” * 4
“testtesttesttest”
```

This doesn’t work

```
>>> “test” * 4.2
```



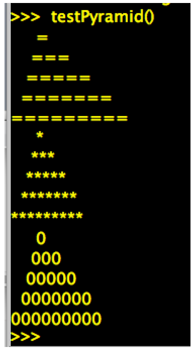
```
>>> "test"*4
'testtesttesttest'
>>> "test"*4.2
The error value is: can't multiply sequence by non-int of type 'float'
Inappropriate argument type.
An attempt was made to call a function with a parameter of an invalid type. This means that you
did something such as trying to pass a string to a method that is expecting an integer.
>>>
```

some more string multiplication

```
def pyramid(character):
   #This function prints a pyramid of the character provided
   space = " " # define a variable - (makes the code clearer)
   print 4 * space, character
   print 3 * space, 3 * character
   print 2 * space, 5 * character
   print space, 7 * character
   print 9 * character

def testPyramid():
    #test the pyramid function with a few different characters
    pyramid("=")
    pyramid("*")
    pyramid("0")
```

some more string multiplication

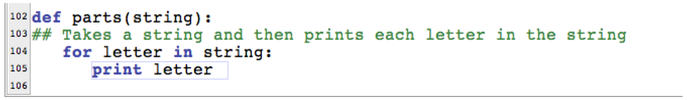
```
def pyramid(character):
   #This function prints a pyramid
   space = " " # define a variable
   print 4 * space, character
   print 3 * space, 3 * character
   print 2 * space, 5 * character
   print space, 7 * character
   print 9 * character

def testPyramid():
    #test the pyramid function with
    pyramid("=")
    pyramid("*")
    pyramid("0")
```



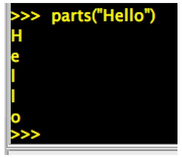
```
>>> testPyramid()
    =
   ===
  =====
 =======
=========
    *
   ***
  *****
 *******
*********
    0
   000
  00000
 0000000
000000000
>>>
```

Taking apart strings

```
def parts(string):
## Takes a string and then prints each letter in the string
   for letter in string
      print letter
```



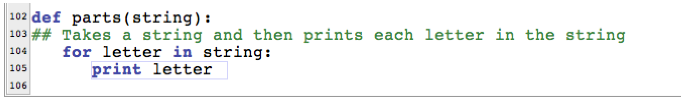
```
def parts(string):
## Takes a string and then prints each letter in the string
   for letter in string:
      print letter
```



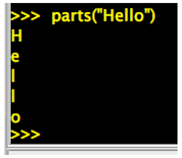
```
>>> parts("Hello")
H
e
l
l
o
>>>
```

Taking apart strings

This is a for loop. Note the “for” word and the “in” word are both keywords defined in python. (Note they are blue in JES – you can’t use keywords for your own variable names)



```
def parts(string):
## Takes a string and then prints each letter in the string
   for letter in string:
      print letter
```



```
>>> parts("Hello")
H
e
l
l
o
>>>
```

Taking apart strings

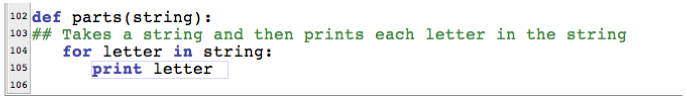
“letter” is an *index* variable that will take on the value of each element of the collection (you decide on what to call this)

The word “in” has to be there – it’s a keyword

The colon (“:”) says, “Next comes the body of the loop.”

The statements in the body of the loop must be indented.

Anything can be inside the for loop – in this case it’s just a single print statement.



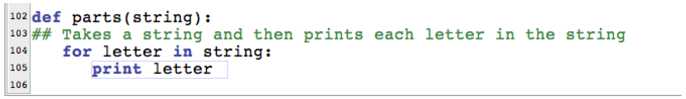
```
def parts(string):
## Takes a string and then prints each letter in the string
    for letter in string:
        print letter
```

Format of the for loop

Notes

1. Computers are not clever – you will need to provide the correct format (syntax) for your *for loops* – or the computer will get upset!

2. Actually this is just one way to write a *for loop* in python – we will see some variations on this during the course.

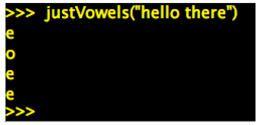


```
def parts(string):
## Takes a string and then prints each letter in the string
    for letter in string:
        print letter
```

Now some selection

Lets just print the vowels in a string – we need a loop and an “if” statement

```
def justVowels(aString):
    ## Takes a string and prints all the vowels in the
    ## string (one on each line)
    for letter in aString:
       if letter in "aeiou":
          print letter
```

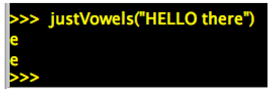


```
>>> justVowels("hello there")
e
o
e
e
>>>
```

One problem

Is “E” a vowel? Well not in our code. We have a semantic error. We are only checking for lowercase “aeiou”

```
def justVowels(aString):
    ## Takes a string and prints all the vowels in the
    ## string (one on each line)
    for letter in aString:
       if letter in "aeiou":
          print letter
```

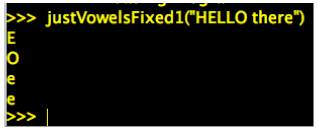


```
>>> justVowels("HELLO there")
e
e
>>>
```

One fix

We can check for lowercase and uppercase “AEIOUaeiou”

```
def justVowelsFixed1(aString):
    ## Takes a string and prints all the vowels in the
    ## string (one on each line)
    for letter in aString:
       if letter in ”AEIOUaeiou":
          print letter
```

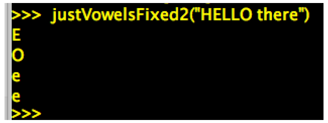


```
>>> justVowelsFixed1("HELLO there")
E
O
e
e
>>> |
```

A better fix ??

We can use the lower() function that will turn a string into all lower case – notice how we call it letter.lower()

```
def justVowelsFixed2(aString):
    ## Takes a string and prints all the vowels in the
    ## string (one on each line)
    for letter in aString:
       if letter.lower() in ”aeiou":
          print letter
```

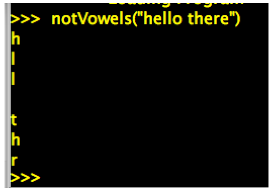


```
>>> justVowelsFixed2("HELLO there")
E
O
e
e
>>>
```

some more selection

Lets print anything that is not a vowel in a string – we need a loop and an “if” statement and we will use the “not” operator

```
def notVowels(aString):
    ## Takes a string and prints all the vowels in the
    ## string (one on each line)
    for letter in aString:
       if not (letter.lower() in "aeiou”):
          print letter
```

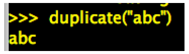


```
>>> notVowels("hello there")
h
l
l
 
t
h
r
>>>
```

and we should fix the case problem here as well…
I’ll leave that to you

duplication

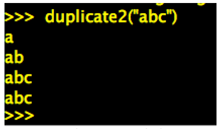
```
def duplicate(sourceString):
    ### This function duplicates the sourceString
    ### It's not a very useful function really -
    ### it just prints the copied string

    duplicateString = ""   # start with an empty string

    # now add each letter in the source string onto the
    # end of the duplicate
    for nextLetter in sourceString:
       duplicateString = duplicateString + nextLetter

    # print the final result - this is outside the for loop
    print duplicateString
```



```
>>> duplicate("abc")
abc
```

duplication – test the loop

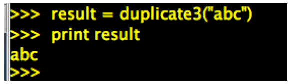
```
def duplicate2(sourceString):
    ### This function duplicates the sourceString
    ### It's not a very useful function really -
    ### it just prints the copied string

    duplicateString = ""   # start with an empty string

    # now add each letter in the source string onto the
    # end of the duplicate
    for nextLetter in sourceString:
       duplicateString = duplicateString + nextLetter
       print duplicateString # this print is inside the for loop

    # print the final result - this is outside the for loop
    print duplicateString
```



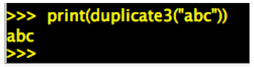
```
>>> duplicate2("abc")
a
ab
abc
abc
>>>
```

duplication – return the result

```
def duplicate3(sourceString):
    ### This function duplicates the sourceString
    ### and returns the result

    duplicateString = ""   # start with an empty string

    # now add each letter in the source string onto the
    # end of the duplicate
    for nextLetter in sourceString:
       duplicateString = duplicateString + nextLetter

    # return the final result -
    # this lets it be used outside the function itself
    return duplicateString
```



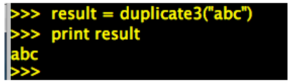
```
>>> result = duplicate3("abc")
>>> print result
abc
>>>
```



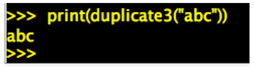
```
>>> print(duplicate3("abc"))
abc
>>>
```

duplication – return the result

```
def duplicate3(sourceString):
    ### This function duplicates the sourceString
    ### and returns the result

    duplicateString = ""   # start with an empty string

    # now add each letter in the source string onto the
    # end of the duplicate
    for nextLetter in sourceString:
       duplicateString = duplicateString + nextLetter

    # return the final result -
    # this lets it be used outside the function itself
    return duplicateString
```



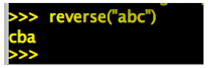
```
>>> result = duplicate3("abc")
>>> print result
abc
>>>
```



```
>>> print(duplicate3("abc"))
abc
>>>
```

Reverse – more interesting ?

```
def reverse(sourceString):
    ### This function prints the reverse of the sourceString

    reverseString = ""   # start with an empty string

    # now add each letter in the source string onto the
    # end of the duplicate
    for nextLetter in sourceString:
       reverseString = nextLetter + reverseString

    # print the final result -
    return reverseString
```



```
>>> reverse("abc")
cba
>>>
```

Strings and Index

```
phrase = "hello World"
```

phrase	H	e	l	l	o		W	o	r	l	d	
	0	1	2	3	4	5	6	7	8	9	10	index

A string is stored in successive memory location characters, with each character having its own index

```
 phrase[0]   is  "H"
 phrase[4]   is  "o"
 phrase[5]   is  " "
phrase[10]  is  "d"
phrase[-1]  is  "d"
phrase[-3]  is  "r"
phrase[11]  does not exist!
phrase[-12]  does not exist!
```

Strings and Index

```
phrase = "hello World"
```

phrase	H	e	l	l	o		W	o	r	l	d	
	0	1	2	3	4	5	6	7	8	9	10	index

A string is stored in successive memory location characters, with each character having its own index

```
 phrase[11]  does not exist!
phrase[-12]  does not exist!
```

Common Bug: The Length Is One More than the Last Index

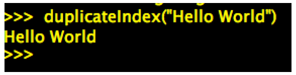
The most common bug in indexing is forgetting that the first index is zero. Because it's zero, the *length* of the string is *one more than* the last index in the string. The last index in a string is length minus one. If you try to get the character beyond the last index, you will get a sequence index error.

Another Duplicate with index

```
phrase = "hello World"
```

phrase	H	e	l	l	o		W	o	r	l	d	
	0	1	2	3	4	5	6	7	8	9	10	index

```
def duplicateIndex(sourceString):

    duplicateString = ""
    numberLetters = 11; # for 11 letters - eg "hello World"

    # only works if the sourceString has 11 letters
    for index in range(0,numberLetters):
       duplicateString = duplicateString + sourceString[index]

    print duplicateString
```



```
>>> duplicateIndex("Hello World")
Hello World
>>>
```

Great if you have 11 characters

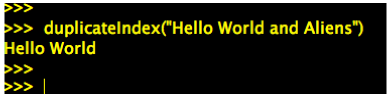
Mod2_2_QuickStart.py

That index bug

```
phrase = "hello World"
```

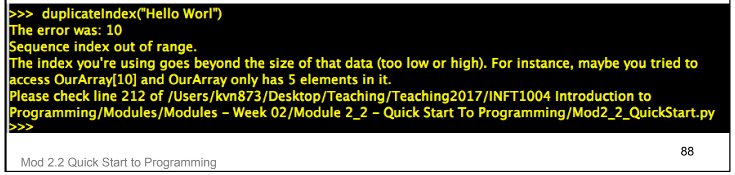
phrase	H	e	l	l	o		W	o	r	l	d	
	0	1	2	3	4	5	6	7	8	9	10	index

More than 11 characters



```
>>>
>>> duplicateIndex("Hello World and Aliens")
Hello World
>>>
>>> |
```

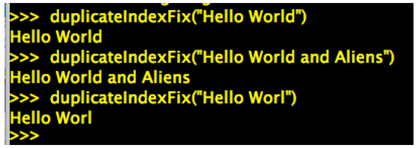
Less than 11 characters



```
>>> duplicateIndex("Hello Worl")
The error was: 10
Sequence index out of range.
The index you're using goes beyond the size of that data (too low or high). For instance, maybe you tried to
access OurArray[10] and OurArray only has 5 elements in it.
Please check line 212 of /Users/kvn873/Desktop/Teaching/Teaching2017/INFT1004 Introduction to
Programming/Modules/Modules - Week 02/Module 2_2 - Quick Start To Programming/Mod2_2_QuickStart.py
>>>
```

Fixing Duplicate with len()

```
def duplicateIndexFix(sourceString):

    duplicateString = "”

    #len(sourceString) returns the number of letters
    numberLetters = len(sourceString);

    for index in range(0,numberLetters):
       duplicateString = duplicateString + sourceString[index]

    print duplicateString
```



```
>>> duplicateIndexFix("Hello World")
Hello World
>>> duplicateIndexFix("Hello World and Aliens")
Hello World and Aliens
>>> duplicateIndexFix("Hello Worl")
Hello Worl
>>>
```

Works with any number of characters in the string

Mod2_2_QuickStart.py

The range function

This is a very useful function for creating a list of numbers

(you will find it works great if you want to go through items in a list – using the index – we will reuse it a lot)

```
def tryRange():
   # some examples of using the range function
   # If you don't provide three arguments, it uses '1'

   print range(0,10)    # [0,1,2,3,4,5,6,7,8,9]  - no 10
   print range(5,10)    # [5,6,7,8,9]
   print range(0,10,2)  # [0,2,4,6,8]
   print range(10,0,-1) # [10,9,8,7,6,5,4,3,2,1] - no 0
```

Mod2_2_QuickStart.py

Familiar Patterns

To be a programmer you need to learn to solve problems using some recurring patterns:

1. You need to store data with a name(s). (variables)
2. Create a sequence of instructions, that the computer can follow. (sequence, functions, program)
3. Use data structures (lists, arrays).
4. Transform data into other forms.
5. Create a set of instructions that are repeated a number of times. (loops, iteration)
6. How to test data (is true or not?), then take actions depending on what the result is. (selection, booleans)

INFT1004

Introduction to Programming

Module 2.3
Programming Style

Comments in code

Round about now, bits of our programs are reaching the point where we might have trouble reading and understanding them

Comments in code are really helpful in this regard

A comment in the code begins with the # symbol

Python just ignores it; it's there for the people who read the program, to help them understand it

Comments in code

Even if you understand a program when you write it, you can have trouble understanding it a few weeks later

So even if you think you're the only one who will ever read your code, include comments in it

Revision - Comments

Your programs must include three kinds of comments:

1. A comment at the start of every program, saying who wrote it, and when, and why
2. A comment at the start of every function, explaining briefly what it does
3. A comment with every bit of code that another programmer might find easier to understand if it's explained

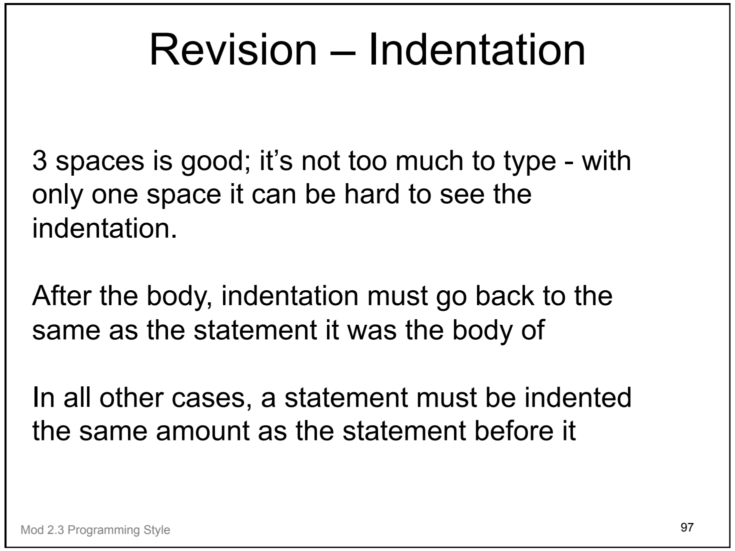
> Try not to write comments to explain what will be obvious to a reasonable programmer!

Indentation

Indentation – how far across the 'page' each statement starts – is absolutely integral to Python programming

Statements that have 'bodies' (eg def, if, for) end with colons

The body of a statement must be indented further than the statement itself

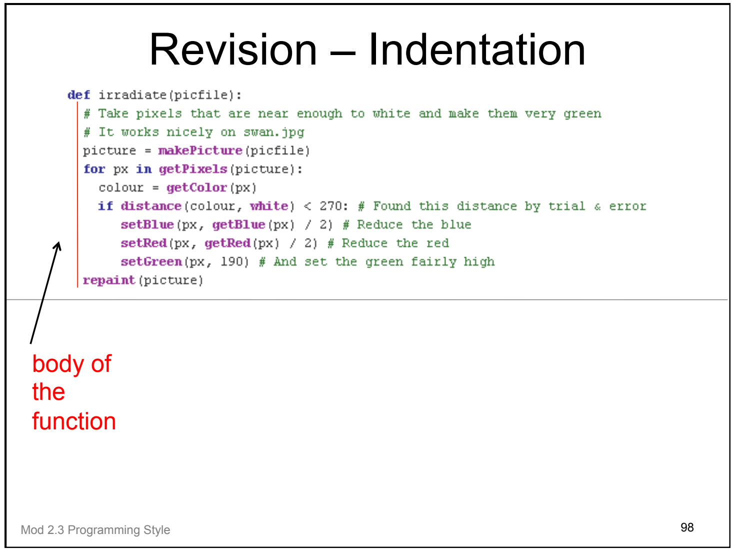


Revision – Indentation

3 spaces is good; it’s not too much to type - with only one space it can be hard to see the indentation.

After the body, indentation must go back to the same as the statement it was the body of

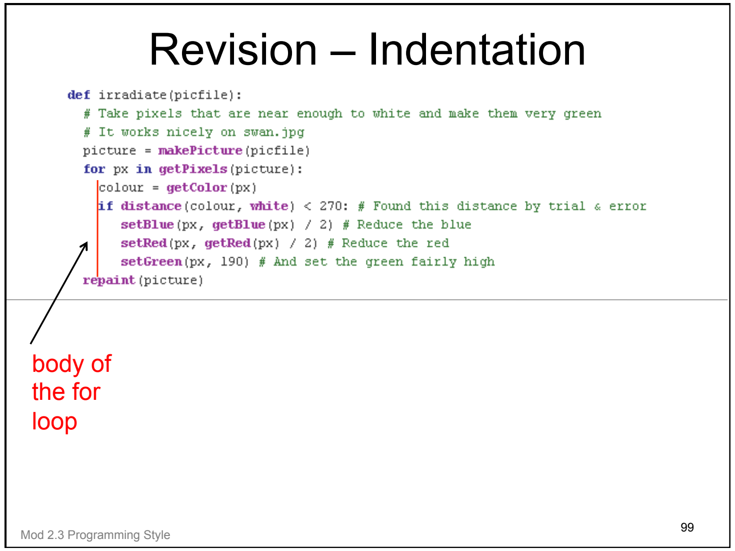
In all other cases, a statement must be indented the same amount as the statement before it



Revision – Indentation

```
def irradiate(picfile):
  # Take pixels that are near enough to white and make them very green
  # It works nicely on swan.jpg
  picture = makePicture(picfile)
  for px in getPixels(picture):
    colour = getColor(px)
    if distance(colour, white) < 270: # Found this distance by trial & error
       setBlue(px, getBlue(px) / 2) # Reduce the blue
       setRed(px, getRed(px) / 2) # Reduce the red
       setGreen(px, 190) # And set the green fairly high
  repaint(picture)
```

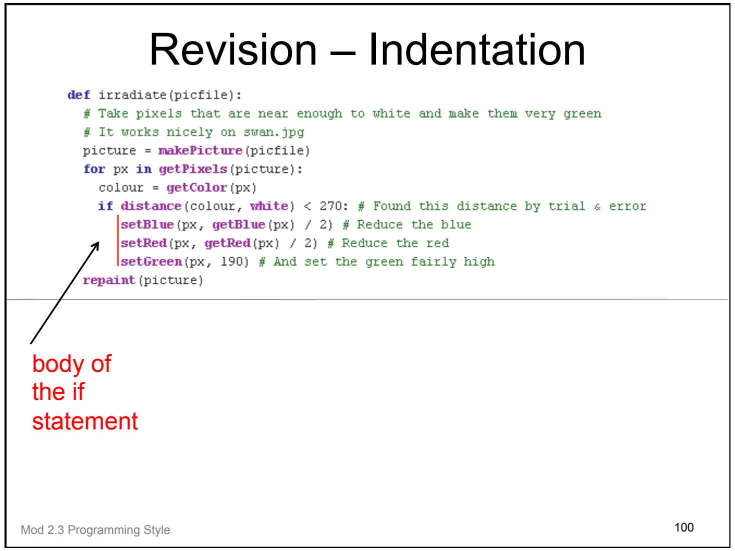
body of the function



Revision – Indentation

```
def irradiate(picfile):
  # Take pixels that are near enough to white and make them very green
  # It works nicely on swan.jpg
  picture = makePicture(picfile)
  for px in getPixels(picture):
    colour = getColor(px)
    if distance(colour, white) < 270: # Found this distance by trial & error
       setBlue(px, getBlue(px) / 2) # Reduce the blue
       setRed(px, getRed(px) / 2) # Reduce the red
       setGreen(px, 190) # And set the green fairly high
  repaint(picture)
```

body of the for loop

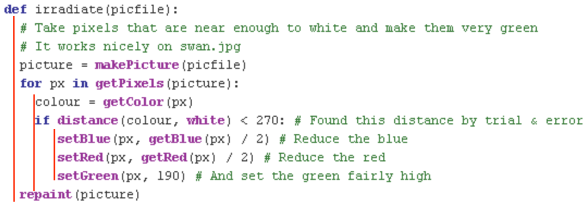


Revision – Indentation

```
def irradiate(picfile):
  # Take pixels that are near enough to white and make them very green
  # It works nicely on swan.jpg
  picture = makePicture(picfile)
  for px in getPixels(picture):
    colour = getColor(px)
    if distance(colour, white) < 270: # Found this distance by trial & error
       setBlue(px, getBlue(px) / 2) # Reduce the blue
       setRed(px, getRed(px) / 2) # Reduce the red
       setGreen(px, 190) # And set the green fairly high
  repaint(picture)
```

body of the if statement

Revision – Indentation



```
def irradiate(picfile):
  # Take pixels that are near enough to white and make them very green
  # It works nicely on swan.jpg
  picture = makePicture(picfile)
  for px in getPixels(picture):
    colour = getColor(px)
    if distance(colour, white) < 270: # Found this distance by trial & error
      setBlue(px, getBlue(px) / 2) # Reduce the blue
      setRed(px, getRed(px) / 2) # Reduce the red
      setGreen(px, 190) # And set the green fairly high
  repaint(picture)
```

Indentation is not just pretty, not just arbitrary

It is what tells Python (and readers) the structure of the program.

For the sake of readers, it helps if the comments are indented the same as their surrounding statements

Names

It's important to choose names that help us, the readers, know what they refer to

The **names** we make up should be *meaningful* and *informative*

Avoid abbreviation in names (the time you save typing a few characters will be lost trying to remember what the abbreviations stand for

Naming Standard – Variables

Camel Case (Variables)

first word is in lowercase, every new word starts in upper case and rest of word is in lower case.

```
heightGirls

finished

fastLap
```

Naming Standard – Functions

Camel Case (Functions and Parameters)

first word is in lowercase, every new word starts in upper case and rest of word is in lower case.

```
calculateArea(height, baseWidth)

calculateMean(listNumbers)

showTreePositions()
```

Naming Standard – Constants

Camel Case (CONSTANTS)

Each word is completely in upper case.
Underscore is used to separate words

```
MAX_HEIGHT

PI

TAX_RATE
```

Naming Standard – Class Names

Pascal Case (Class names)

Each word (including the first begins with an upper case character. All other characters in the word are lower case.

```
Circle

UniversityStudent

LinearList
```

Caution - Choosing names

Words in magenta are words that have some special meaning in JES

You choose a name for a variable. When you type it, JES makes it magenta. Then you choose a bad name for your variable. Choose a different name!

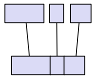
If JES lets you use one as a variable, you will change its meaning! This is something to avoid.

Using functions well

Every function should have just one purpose and do just one thing!

If you want a function to do several things, each of which is clearly defined . . .

1. write a function for each of those things
2. then write another function that calls each of them in turn



This is called 'functional decomposition' . . . taking the task to be done and decomposing it into subtasks, each in its own function

Functions to avoid repeated code

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

Programs - Functions

Programs are made up of one of more functions

If we want a function to take *arguments*, we include corresponding *parameters* when defining it

(We will talk a lot more about defining functions soon)

Functions : Return

We can allow someone to pass information into a function by defining the function to have parameters.

We can also pass information out of a function by using the return statement.

Don't forget it is best practice to only ever have one return statement in a function

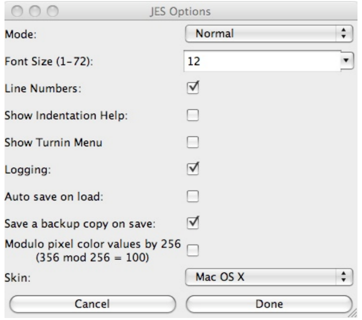
And be careful of side effects with Python parameters that are complex types.

JES Options

Some useful things you can do on the Edit/Options menu

1. Change Mode to Expert – gives more help on errors, which might or might not be more helpful

2. Auto save on load – means you don't have to agree to save every time you load the program



JES Options

Mode: Normal

Font Size (1–72): 12

Line Numbers: ☑

Show Indentation Help: □

Show Turnin Menu □

Logging: ☑

Auto save on load: □

Save a backup copy on save: ☑

Modulo pixel color values by 256 (356 mod 256 = 100) □

Skin: Mac OS X

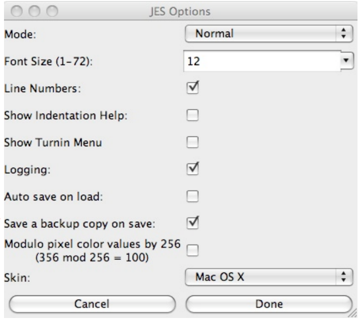
JES Options

Some useful things you can do on the Edit/Options menu

3. Consider turning off logging

4. Consider turning off saving backup copies

5. Look at the modulo option for pixel colour values; you may or may not need this depending on what you are doing



JES Options

Mode: Normal

Font Size (1–72): 12

Line Numbers: ☑

Show Indentation Help: □

Show Turnin Menu □

Logging: ☑

Auto save on load: □

Save a backup copy on save: ☑

Modulo pixel color values by 256 (356 mod 256 = 100) □

Skin: Mac OS X

What to do this week

□ Do the Week 2 labs (bring your problems to class)

□ Read the textbook (Ch 1, 2, 3)