Python

- Python is a computer language (as is Wolfram). You can use it for lots more than mathematics.
- It is named after its inventor, Monty Python (Wolfram is named after its inventor, Stephen Wolfram)
- PyCharm is an integrated development environment that makes Python more enjoyable to code
- Sage is a maths environment rather like Mathematica where you can do maths using Python.
- SageMathCloud is a web service that runs Sage for you `https://cloud.sagemath.com`

Using Python Interactively

- Open PyCharm
- Create a project called PythonLecture1
- Select the unlabelled button in the bottom left hand corner (obviously)
- Select **Python console**
- Run the following commands

```
a = 3
b = 4
c = a**2 + b**2
print(c)
a,b = 5,12
print(c)
```

Writing a Python script

- Right click on “PythonLecture1”, choose **New**
- Select **Python file**
- Call it `triads.py`
- Insert the following code, then right click the file and **Run** it

```python
for m in range(1,100):
    for n in range(1,m):
        a = m**2 - n**2
        b = 2*m*n
        c = m**2 + n**2
        assert a**2 + b**2 == c**2
        print( "m="+str(m)+", n="+str(n) )
        print( "Triple "+str(a)+", "+str(b)+", "+str(c) )
print('That\'s enough')
```

Observations

- `range(1,10)` starts at 1 and ends at 9
- You need to type `*` for multiplication
- `==` tests equality
- `assert` means much the same as in Mathematica
- You can use `"` to create strings
- You can use `+` to concatentate strings
- You us `for` with `in` and don't forget the `:`
- You group code using tabs (which should be 4 characters wide)
- You can use ‘ to create strings too
- You can use \ to *escape* special characters
- With scripts your code is saved. The console is interactive.
- Use round brackets to call functions

Maths functions

To use basic maths functions you can do any one of the following

```
import math
root2 = math.sqrt(2)
```

```
import math as m
root2 = m.sqrt(2)
```

```
from math import sqrt
root2 = sqrt(2)
```

```
from math import *
root2 = sqrt(2)
```

Symbolic calculations

To perform symbolic calculations use the package sympy. First we must install it.

- Select **File->Settings->Project->Project Interpreter**
- Click the +
- Type in sympy and click **Install Package**
- Now try running the following

```
import sympy

x, y, theta = sympy.var(’x y theta’)
x = sympy.cos(theta)
y = sympy.sin(theta)
print( sympy.simplify( x**2 + y**2 ))
```

Observations

- There is more than one `sin` function in Python, a numerical one and a symbolic one.
- You must use `sympy.var` to declare which variables should be treated symbolically.
- Question: how would you avoid typing `sympy` so often?
- Question: why aren't `math` and `sympy` automatically imported?
- Some functions, such as `var` appear to return multiple values.

Tuples

- A tuple is an *immutable* data structure consisting of a number of elements
- 1,2,4,8 is a tuple of four elements
- () is the empty tuple
- (7,) is a tuple of length 1
- Use [] to access elements of a tuple, starting at 0

```
triple = 3, 4, 5
assert triple[0]**2 + triple[1]**2==triple[2]**2
emptyTuple = ()
assert len( emptyTuple )==0
tripleOfTriples= (3,4,5),(5,12,13),(9,40,41)
#triple[2]=7
singlet="vest", #try removing the comma
len(singlet)
```

Lists

- A list is a *mutable* data structure consisting of a number of elements
- `[1,2,4,8]` is a list of four elements
- `[]` is the empty list
- `[1]` is a list of one elements
- Use `append` to add to a list
- Use `[]` to access elements of a list, starting at 0

```
squares = []
for i in range(1,100):
    squares.append(i**2)
s = 0
for i in range(0,len(squares)):
    s = s+squares[i]
print( s)
```

Iterating

Note that we started at 0 and ended at `len` when looping. Here's a better approach.

```
s = 0
for square in squares:
    s += square
print( s)
```

Note the `+=`. This is often quite convenient.

```
soliloquoy = """HAMLET: To be, or not to be--that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune
Or to take arms against a sea of troubles
And by opposing end them."""
for c in soliloquoy:
    print (c)
```

Slicing

Slicing strings

```
str = "0123456789"
print(str[7])
print(str[1:8])
print(str[1:-1])
print(str[1:])
print(str[:8])
print(str[:])
```

Slicing lists

```
vec = [0,1,2,3,4,5,6,7,8,9]
print(vec[7])
print(vec[1:8])
vec[3:9] = ["..."]
print(vec)
```

Slicing

```
   S   l   i   c   e   M   e
 0   1   2   3   4   5   6   7
-7  -6  -5  -4  -3  -2  -1
```

```
str = "SliceMe"
print( str[0:5] )
print( str[3:0] )
```

Functions

A function to numerically solve a quadratic equation

```
def solve_quadratic(a,b,c):
    discriminant = b**2-4*a*c
    x1 = (-b + math.sqrt(discriminant))/(2*a)
    x2 = (-b - math.sqrt(discriminant))/(2*a)
    return x1,x2

a = 2; b = 3; c = -7
y1,y2 = solve_quadratic(a,b,c)
print(’Solutions are {} and {}’.format(y1,y2))
for x in y1,y2:
    print(a*x*x+b*x+c)
```

Using nosetests with PyCharm

- Install the package `nose`
- Select **File->Settings->Tools->Python Integrated Python Tools->Default Test Runner->nosetests**

Nose is a package which makes it easy to test your Python code. Actually, one should probably use nose2 these days, but there aren't any significant differences between different Python testing packages, so I haven't updated these slides.

Unit tests

In file mymath

```
def solve_quadratic(a,b,c):
    discriminant = b**2-4*a*c
    x1 = (-b + math.sqrt(discriminant))/(2*a)
    x2 = (-b - math.sqrt(discriminant))/(2*a)
    return x1,x2
```

In file mymath_tests

```
import mymath
import nose.tools

def test_solve_quadratic():
    a = 2; b = 3; c = -7
    x1,x2 = mymath.solve_quadratic(a,b,c)
    for x in x1,x2:
        nose.tools.assert_almost_equals(a * x * x + b * x + c, 0.0)
```

Unit tests

- The single biggest idea in computer programming of the 1990s
- Took a decade (or more) to fully catch on
- All your code should be tested
- All tests must be fully automated
- All your tests should be run regularly at the click of a button
- Write small functions with tests
- Any well-designed code should be testable. That is part of what well-designed means.
- A *unit test* tests a small piece of code such as a single function
- A *system test* tests the whole software system
- A *smoke test* tests things superficially work OK
- Human beings are unreliable and expensive.

Be test-infected

- Write your tests before you write your code
- If you ever detect a bug in your code, write a test that identifies the bug so it can never happen again
- Don't write scripts, write tests

If statements

```
def victor(x,y):
    """Return the index of the victor, or None"""
    if x=='paper':
        if y=='paper':
            return None
        elif y=='scissors':
            return 1
        elif y=='stone':
            return 0
        else:
            raise Exception('Invalid value '+str(y))
    elseif x=='scissors':
    # you get the picture
    # ...
```

- We have a docstring describing what the function does. Click ctrl and hover over a function call to see the docstring.
- You can generate errors with `raise Exception`. Don't just print things out!
- There is a special data item called None

Logical operators

```
def victor(x,y):
    if (x==’paper’ and y==’paper’) or \
       (x==’stone’ and y==’stone’) or \
       (x==’scissors’ and y==’scissors’):
        return None
    # you get the picture
```

This example shows that you can break a statement up over multiple lines using \

While statements

```python
def is_fibonacci( x ):
    n=1
    fib = 0
    while fib<x:
        fib = fibonacci(n)
        if fib==x:
            return True
        n += 1
    return False
```

- A while loop continues until the test statement is False
- True and False

Miscellany

- The function `abs` computes the absolute value
- The symbol `%` means modulo. `print(7 %3)`
- The symbol // means flor division. `print(7//3)`
- The function `math.floor` computes the integer below
- The function `math.ceil` computes the integer above
- PyCharm will auto complete for you when you type `math.`

Exercises

Put all your answers in `mymath.py` or `mymath_tests.py`

1. Write a function `fibonacci` that returns the n-th Fibonacci number
2. **Write a test for this function**
3. Write a function `fibonacciNumbers` that returns the first n Fibonacci numbers
4. **Write a test for this function**
5. Write a function that computes the greatest common divisor of two integers a and b
6. **Write a test for this function**
7. Write a function that allows you to find x and y such that $xa + yb = \gcd(a, b)$.
8. **Guess what question you are being asked. Answer it.**
9. What is wrong with the ordering of these questions?