

# MAT 305: Mathematical Computing

## Decision-making

John Perry

University of Southern Mississippi

Spring 2017

# Outline

- 1 Decision-making
  - Concavity
  - Method of bisection
  - Piecewise function
- 2 Boolean statements
- 3 Having said all that...
- 4 Breaking loops
- 5 Raising and handling exceptions
- 6 Summary

Decision-  
making

Concavity  
Method of bisection  
Piecewise function

Boolean  
statements

Having said all  
that...

Breaking loops

Raising and  
handling  
exceptions

Summary

# Outline

- 1 Decision-making
  - Concavity
  - Method of bisection
  - Piecewise function
- 2 Boolean statements
- 3 Having said all that...
- 4 Breaking loops
- 5 Raising and handling exceptions
- 6 Summary

Decision-  
making

Concavity

Method of bisection

Piecewise function

Boolean  
statements

Having said all  
that...

Breaking loops

Raising and  
handling  
exceptions

Summary

# Decision making?

A function may have to act in different ways, depending on the arguments.

## Decision making?

A function may have to act in different ways, depending on the arguments.

### Example

Piecewise functions:

$$f(x) = \begin{cases} f_1(x), & x \in (a_0, a_1) \\ f_2(x), & x \in [a_1, a_2). \end{cases}$$

If  $x \in (a_0, a_1)$ , then  $f(x) = f_1(x)$ ;  
if  $x \in [a_1, a_2)$ , then  $f(x) = f_2(x)$ .

## Decision making?

A function may have to act in different ways, depending on the arguments.

### Example

Deciding concavity:

If  $f''(a) > 0$ , then  $f$  is concave up at  $x = a$ ;  
if  $f''(a) < 0$ , then  $f$  is concave down at  $x = a$ .

# Method of bisection

## From Calculus I:

- We want to find the root of a *continuous* polynomial.
- We know it lies on somewhere on an interval.
- By repeatedly halving the interval, we can narrow down to something “accurate”

# Pseudocode style

```
repeat  $n$  times  
  let  $c = a+b/2$   
  if  $f(a), f(c)$  same sign  
    replace  $a$  by  $c$   
  otherwise  
    replace  $b$  by  $c$ 
```



## if statements

```
if condition :  
    if-statement1  
    if-statement2  
    ...  
non-if statement1
```

where

- *condition*: expression that evaluates to True or False
- *condition* True? *if-statement1*, *if-statement2*, ... performed
  - proceed eventually to *non-if statement1*
- *condition* False? *if-statement1*, *if-statement2*, ... skipped
  - proceed immediately to *non-if statement1*

## Example

```
sage: f(x) = cos(x)
```

```
sage: ddf(x) = diff(f,2)
```

```
sage: if ddf(3*pi/4) > 0:  
      print 'concave up at', 3*pi/4  
concave up at 3/4*pi
```

## if-else statements

if *condition*:  
    *if-statement1*

    ...

else:  
    *else-statement1*

    ...

*non-if statement1*

where

- *condition* True? *if-statement1*, ... performed
  - *else-statement1*, ... skipped
- *condition* False? *else-statement1*, ... performed
  - *statement1*, ... skipped
- proceed sooner or later to *non-if statement1*

## if-elif-else statements

```
if condition1:  
    if-statement1  
    ...  
elif condition2:  
    elif1-statement1  
    ...  
elif condition3:  
    elif2-statement1  
    ...  
...  
else:  
    else-statement1  
    ...  
non-if statement1
```

## Pseudocode for `if-elif-else`

```
if condition1  
    if-statement1  
    ...  
else if condition2  
    elseif1-statement1  
    ...  
else if condition3  
    elseif2-statement1  
    ...  
...  
else  
    else-statement1  
    ...
```

Notice:

- indentation
- no colons
- **else if**, not **elif**

## Example: concavity

Write a Sage function that tests whether a function  $f$  is concave up or down at a given point. Have it return the string 'concave up', 'concave down', or 'neither'.

## Example: concavity

Write a Sage function that tests whether a function  $f$  is concave up or down at a given point. Have it return the string 'concave up', 'concave down', or 'neither'.

Different choices  $\implies$  need to decide!  $\implies$  **if**

## Example: concavity

Write a Sage function that tests whether a function  $f$  is concave up or down at a given point. Have it return the string 'concave up', 'concave down', or 'neither'.

Different choices  $\implies$  need to decide!  $\implies$  **if**

Start with pseudocode.

- inputs needed?
- output expected?
- what to do?
  - step by step
  - *Divide et impera!* Divide and conquer!



# Pseudocode for Example

**algorithm** *check\_concavity*

**inputs**

# Pseudocode for Example

**algorithm** *check\_concavity*

**inputs**

$$a \in \mathbb{R}$$

$f(x)$ , a twice-differentiable function at  $x = a$

**outputs**

## Pseudocode for Example

**algorithm** *check\_concavity*

**inputs**

$$a \in \mathbb{R}$$

$f(x)$ , a twice-differentiable function at  $x = a$

**outputs**

'concave up' if  $f$  is concave up at  $x = a$

'concave down' if  $f$  is concave down at  $x = a$

'neither' otherwise

**do**

## Pseudocode for Example

**algorithm** *check\_concavity*

**inputs**

$a \in \mathbb{R}$

$f(x)$ , a twice-differentiable function at  $x = a$

**outputs**

'concave up' if  $f$  is concave up at  $x = a$

'concave down' if  $f$  is concave down at  $x = a$

'neither' otherwise

**do**

**if**  $f''(a) > 0$

**return** 'concave up'

**else if**  $f''(a) < 0$

**return** 'concave down'

**else**

**return** 'neither'

Try it!

```
sage: def check_concavity(a, f, x):  
      ddf = diff(f, x, 2)  
      if ddf(x=a) > 0:  
          return 'concave up'  
      elif ddf(x=a) < 0:  
          return 'concave down'  
      else:  
          return 'neither'
```

Decision-  
making

Concavity

Method of bisection

Piecewise function

Boolean  
statements

Having said all  
that...

Breaking loops

Raising and  
handling  
exceptions

Summary

Try it!

```
sage: def check_concavity(a, f, x):
      ddf = diff(f, x, 2)
      if ddf(x=a) > 0:
          return 'concave up'
      elif ddf(x=a) < 0:
          return 'concave down'
      else:
          return 'neither'

sage: check_concavity(3*pi/4, cos(x), x)
'concave up'

sage: check_concavity(pi/4, cos(x), x)
'concave down'
```

## More interesting example

Use the Method of Bisection to approximate a root of  $\cos x - x$  on the interval  $[0, 1]$ , correct to the hundredths place.

## More interesting example

Use the Method of Bisection to approximate a root of  $\cos x - x$  on the interval  $[0, 1]$ , correct to the hundredths place.

???



# Method of Bisection?

The Method of Bisection is based on:

## Theorem (Intermediate Value Theorem)

*If*

- *$f$  is a continuous function on  $[a, b]$ , and*
- *$f(a) \neq f(b)$ ,*

*then*

- *for any  $y$  between  $f(a)$  and  $f(b)$ ,*
- *$\exists c \in (a, b)$  such that  $f(c) = y$ .*

# Continuous?

$f$  *continuous* at  $x = a$  if

- can evaluate limit at  $x = a$  by computing  $f(a)$ , or
- can draw graph without lifting pencil

# Continuous?

$f$  continuous at  $x = a$  if

- can evaluate limit at  $x = a$  by computing  $f(a)$ , or
- can draw graph without lifting pencil

**Upshot:** To find a root of a continuous function  $f$ , start with two  $x$  values  $a$  and  $b$  such that  $f(a)$  and  $f(b)$  have different signs, then bisect the interval.

# Method of bisection

**algorithm** *Method\_of\_Bisection*

**inputs**

$$a, b \in \mathbb{R}$$

$f$ , a continuous function on  $[a, b]$

$n$ , number of bisections

**outputs**

$[c, d] \subseteq [a, b]$  such that

$$d - c = (b - a) / 2^n, \text{ and}$$

a root of  $f$  lies in  $[c, d]$

**do**

let  $c = a, d = b$

**repeat**  $n$  times

let  $e = (c + d) / 2$

**if**  $f(c)$  and  $f(e)$  have same sign

replace  $c$  by  $e$

**else**

replace  $d$  by  $e$

**return**  $[c, d]$

## Sage code

```
def method_of_bisection(a, b, n, f, x=x):  
    c, d = a, b  
    f(x) = f  
    for each in xrange(n):  
        e = (c + d) / 2  
        if f(c)*f(e) > 0:  
            c = e  
        else:  
            d = e  
    return (c,d)
```

## Sage code

```
def method_of_bisection(a, b, n, f, x=x):
    c, d = a, b
    f(x) = f
    for each in xrange(n):
        e = (c + d) / 2
        if f(c)*f(e) > 0:
            c = e
        else:
            d = e
    return (c,d)

sage: method_of_bisection(0, 1, 20, cos(x) - x)
(0, 1/1048576)
```

## A more complicated example

How do we handle a piecewise function defined over more complicated intervals?

### Example

Suppose

$$g(x) = \begin{cases} 3x, & x \in [0, 2) \\ -\frac{x}{3} + \frac{20}{3}, & x \in [2, 20) \\ 0, & x \geq 20. \end{cases}$$

How do we define this in Sage?

# Pseudocode deceptively easy

Decision-  
making

Concavity

Method of bisection

Piecewise function

Boolean  
statements

Having said all  
that...

Breaking loops

Raising and  
handling  
exceptions

Summary

**algorithm** *piecewise\_g*

**inputs**

$$a \in [0, \infty)$$

**outputs**

$g(a)$ , where  $g$  is defined as above

**do**

**if**  $a \in [0, 2)$

**return**  $3a$

**else if**  $a \in [2, 20)$

$$\mathbf{return} -\frac{a}{3} + \frac{20}{3}$$

**else**

**return** 0



## Pseudocode deceptively easy

**algorithm** *piecewise\_g*

**inputs**

$$a \in [0, \infty)$$

**outputs**

$g(a)$ , where  $g$  is defined as above

**do**

**if**  $a \in [0, 2)$

**return**  $3a$

**else if**  $a \in [2, 20)$

**return**  $-\frac{a}{3} + \frac{20}{3}$

**else**

**return** 0

...but how does Sage decide  $a \in [x_1, x_2)$ ?!?

# Outline

- ① Decision-making
  - Concavity
  - Method of bisection
  - Piecewise function
- ② Boolean statements
- ③ Having said all that...
- ④ Breaking loops
- ⑤ Raising and handling exceptions
- ⑥ Summary

# Boolean algebra

Boolean algebra operates on only two values:  $\{\text{True}, \text{False}\}$ .

... or  $\{1, 0\}$  if you prefer

... or  $\{\text{Yes}, \text{No}\}$  if you prefer

## Boolean algebra

Boolean algebra operates on only two values: {True, False}.

... or {1, 0} if you prefer

... or {Yes, No} if you prefer

Basic operations:

- **not**  $x$ 
  - True iff  $x$  is False
- $x$  **and**  $y$ 
  - True iff both  $x$  and  $y$  are True
- $x$  **or**  $y$ 
  - True iff
    - $x$  is True; or
    - $y$  is True; or
    - both  $x$  and  $y$  are True

(“inclusive” or)

## Example: and, or

sage:  $5 > 4$

True

obvious enough

sage:  $5 < 4$

False

sage:  $(5 > 4)$  or  $(5 < 4)$

True

because at least one is True ( $5 > 4$ )

sage:  $(5 > 4)$  and  $(5 < 4)$

False

because one is False

## Example: not

```
sage: 4 > 4
```

```
False
```

obvious enough

```
sage: not (4 > 4)
```

```
True
```

```
sage: not ((5 > 4) or (4 < 5))
```

```
False
```

we have (not True)

```
sage: not (4 == 5)
```

```
True
```

we have (not False)

# Equality and inequalities

Recall:  $=$  and  $==$  are not the same

- $x = y$  assigns value of  $y$  to  $x$
- $x == y$  compares values of  $x, y$ , reports True or False

# Equality and inequalities

Recall:  $=$  and  $==$  are not the same

- $x = y$  assigns value of  $y$  to  $x$
- $x == y$  compares values of  $x, y$ , reports True or False

For inequalities,

- $x != y$  compares  $x, y$ 
  - True iff not ( $x == y$ )
- $x > y, x < y$  have usual meanings



# Equality and inequalities

Recall:  $=$  and  $==$  are not the same

- $x = y$  assigns value of  $y$  to  $x$
- $x == y$  compares values of  $x, y$ , reports True or False

For inequalities,

- $x != y$  compares  $x, y$ 
  - True iff not  $(x == y)$
- $x > y, x < y$  have usual meanings
- $x \geq y$ ? use  $x >= y$ 
  - True iff not  $(x < y)$
- $x \leq y$ ? use  $x <= y$ 
  - True iff not  $(x > y)$

## Back to the example

### Example

Suppose

$$g(x) = \begin{cases} 3x, & x \in [0, 2) \\ -\frac{x}{3} + \frac{20}{3}, & x \in [2, 20) \\ 0, & x \geq 20. \end{cases}$$

How do we define this in Sage? Using Boolean algebra, the pseudocode (and Python code) becomes much simpler.

## Pseudocode, again

**algorithm** *piecewise\_g*

**inputs**

$$a \in [0, \infty)$$

**outputs**

$g(a)$ , where  $g$  is defined as above

**do**

**if**  $a \in [0, 2)$

**return**  $3a$

**else if**  $a \in [2, 20)$

**return**  $-\frac{a}{3} + \frac{20}{3}$

**else**

**return**  $0$

## Pseudocode, again

**algorithm** *piecewise\_g*

**inputs**

$$a \in [0, \infty)$$

**outputs**

$g(a)$ , where  $g$  is defined as above

**do**

**if**  $a \in [0, 2)$

**return**  $3a$

**else if**  $a \in [2, 20)$

**return**  $-\frac{a}{3} + \frac{20}{3}$

**else**

**return**  $0$

...but how does does Sage decide  $a \in [x_1, x_2)$ ?!?

use  $a \geq x_1$  **and**  $a < x_2$ !

## Sage code

```
sage: def piecewise_g(a):  
    if (a >= 0) and (a < 2):  
        return 3*a  
    elif (a >= 2) and (a < 20):  
        return -a/3 + 20/3  
    else:  
        return 0
```

## Sage code

```
sage: def piecewise_g(a):  
        if (a >= 0) and (a < 2):  
            return 3*a  
        elif (a >= 2) and (a < 20):  
            return -a/3 + 20/3  
        else:  
            return 0
```

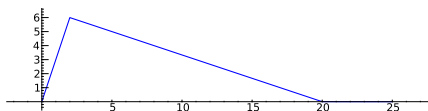
*Much* easier to look at.

You can even test the function

```
sage: def piecewise_g(a): ...
```

```
sage: pgplot = plot(piecewise_g, 0, 25)
```

```
sage: show(pgplot, aspect_ratio=1)
```



Decision-  
making

Concavity

Method of bisection

Piecewise function

Boolean  
statements

Having said all  
that...

Breaking loops

Raising and  
handling  
exceptions

Summary

# Outline

- 1 Decision-making
  - Concavity
  - Method of bisection
  - Piecewise function
- 2 Boolean statements
- 3 Having said all that...
- 4 Breaking loops
- 5 Raising and handling exceptions
- 6 Summary



## There's an error in the code

$$g(x) = \begin{cases} 3x, & x \in [0, 2) \\ -\frac{x}{3} + \frac{20}{3}, & x \in [2, 20) \\ 0, & x \geq 20. \end{cases}$$

What if  $a < 0$ ?

- $g(a)$  undefined, but...
- function returns answer!

```
sage: piecewise_g(-1)
```

```
0
```

Think about

- cause?
- fix?

# Exceptions

One fix is via **exceptions**, the preferred way of dealing with undefined values. Exceptions interrupt a program and propagate an error to the user. We discuss them in some detail later, but here's how you might handle that now:

```
sage: def piecewise_g(a):
        if (a >= 0) and (a < 2):
            return 3*a
        elif (a >= 2) and (a < 20):
            return -a/3 + 20/3
        elif a >= 20:
            return 0
        else:
            raise ValueError, 'The input ' + str(a)
            + ' should be nonnegative.'
```

# Exceptions

One fix is via **exceptions**, the preferred way of dealing with undefined values. Exceptions interrupt a program and propagate an error to the user. We discuss them in some detail later, but here's how you might handle that now:

```
sage: def piecewise_g(a):
        if (a >= 0) and (a < 2):
            return 3*a
        elif (a >= 2) and (a < 20):
            return -a/3 + 20/3
        elif a >= 20:
            return 0
        else:
            raise ValueError, 'The input ' + str(a)
            + ' should be nonnegative.'
```

*Much* easier to look at.

## Sage has a `piecewise()` command...

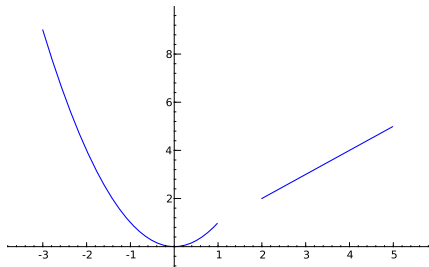
`piecewise([(a1, b1), f1], [(a2, b2), f2], ...])` where

- $a_i, b_i \in \mathbb{R}$
- $f_i$  describes function on interval  $(a_i, b_i)$

...so it's actually a little easier

```
sage: piecewise_g = piecewise([[(-3,1), x**2],  
                               [(2,5), x]])
```

```
sage: plot(piecewise_g, xmin=-3, xmax=3)
```



Decision-  
making

Concavity

Method of bisection

Piecewise function

Boolean  
statements

Having said all  
that...

Breaking loops

Raising and  
handling  
exceptions

Summary

# Outline

- 1 Decision-making
  - Concavity
  - Method of bisection
  - Piecewise function
- 2 Boolean statements
- 3 Having said all that...
- 4 Breaking loops
- 5 Raising and handling exceptions
- 6 Summary

# Don't break what ain't fixed

If a loop arrives at an answer we like, earlier than we expect, we can break out of it.

## Example (Method of bisection)

If we find an *actual root*, no point in continuing the loop.

# Don't break what ain't fixed

If a loop arrives at an answer we like, earlier than we expect, we can break out of it.

## Example (Method of bisection)

If we find an *actual root*, no point in continuing the loop.

`break` tells Sage to break out of the (innermost) loop and continue



```
def method_of_bisection(a, b, n, f, x=x):
    c, d = a, b
    f(x) = f
    for each in xrange(n):
        e = (c + d) / 2
        if f(c)*f(e) > 0:
            c = e
        elif f(e) == 0:
            c = d = e
            break
        else:
            d = e
    return (c,d)
```

Decision-  
making

Concavity

Method of bisection

Piecewise function

Boolean  
statements

Having said all  
that...

Breaking loops

Raising and  
handling  
exceptions

Summary

# Example

```
sage: method_of_bisection(-1, 1, 20, 4*x + 3)
(-3/4, -3/4)
```

## Example

```
sage: method_of_bisection(-1, 1, 20, 4*x + 3)
(-3/4, -3/4)
```

Actual result found in 3 steps, not 20!

$(-1, 1), (-1, 0), (-1, -1/2), (-1, -3/4), (-7/8, -3/4), (-13/16, -3/4),$   
...

# Outline

- 1 Decision-making
  - Concavity
  - Method of bisection
  - Piecewise function
- 2 Boolean statements
- 3 Having said all that...
- 4 Breaking loops
- 5 Raising and handling exceptions
- 6 Summary

# Exceptions

Sage classifies errors according to certain types

- `SyntaxError`
- `TypeError`
- `ZeroDivisionError`

...

# Exceptions

Sage classifies errors according to certain types

- `SyntaxError`
- `TypeError`
- `ZeroDivisionError`

...

The management of special errors is called **exception handling**.

## try/except

Think an exception can occur? use a try/except block.

```
sage: try:  
        a = 1/0  
    except ZeroDivisionError:  
        a = Infinity
```

```
sage: a  
+Infinity
```

## try/except

Think an exception can occur? use a try/except block.

```
sage: try:
        a = 1/0
    except ZeroDivisionError:
        a = Infinity
```

```
sage: a
```

```
+Infinity
```

Need not specify exception type.

```
sage: try:
        a = 1/0
    except:
        a = Infinity
```

```
sage: a
```

```
+Infinity
```



# General form

```
try:  
    try_statement_1  
    try_statement_2  
    ...
```

# General form

```
try:  
    try_statement_1  
    try_statement_2  
    ...  
except exception_1:  
    exception1_statement_1  
    exception1_statement_2  
    ...
```

## General form

```
try:  
    try_statement_1  
    try_statement_2  
    ...  
except exception_1:  
    exception1_statement_1  
    exception1_statement_2  
    ...  
except exception_2:  
    exception2_statement_1  
    exception2_statement_2  
    ...
```

# General form

```
try:  
    try_statement_1  
    try_statement_2  
    ...  
except exception_1:  
    exception1_statement_1  
    exception1_statement_2  
    ...  
except exception_2:  
    exception2_statement_1  
    exception2_statement_2  
    ...  
...
```

## Example

A while back you had to implement a *normal line*. One way to do it:

```
sage: def normal_line(a, f, x=x):  
      f(x) = f  
      df(x) = diff(f)  
      m=1/df(a)  
      return m*(x - a) + f(a)
```

## Example

A while back you had to implement a *normal line*. One way to do it:

```
sage: def normal_line(a, f, x=x):  
      f(x) = f  
      df(x) = diff(f)  
      m=1/df(a)  
      return m*(x - a) + f(a)
```

This has an “obvious” error. What is it?

## Example

A while back you had to implement a *normal line*. One way to do it:

```
sage: def normal_line(a, f, x=x):  
      f(x) = f  
      df(x) = diff(f)  
      m=1/df(a) # Possible division by 0  
      return m*(x - a) + f(a)
```

This has an “obvious” error. What is it?

# Catch, handle the error

Problem arises when  $\Delta x = 0$ :



# Catch, handle the error

Problem arises when  $\Delta x = 0$ :

- implies vertical line

# Catch, handle the error

Problem arises when  $\Delta x = 0$ :

- implies vertical line
- has form  $x = a$

# Catch, handle the error

Problem arises when  $\Delta x = 0$ :

- implies vertical line
- has form  $x = a$
- return this!

## In code

```
sage: def normal_line(a, f, x=x):  
      f(x) = f  
      df(x) = diff(f)  
      try:  
          m = 1/df(a)  
          result = y == m*(x - a) + f(a)  
      except ZeroDivisionError:  
          result = x == a  
      return result
```

## In code

```
sage: def normal_line(a, f, x=x):
      f(x) = f
      df(x) = diff(f)
      try:
          m = 1/df(a)
          result = y == m*(x - a) + f(a)
      except ZeroDivisionError:
          result = x == a
      return result
sage: normal_line(0, x**2)
x == 0
```

# This can raise an error, too

```
sage: normal_line(1, x**2)
```

```
NameError: global name 'y' is not defined
```

# This can raise an error, too

```
sage: normal_line(1, x**2)
```

```
NameError: global name 'y' is not defined
```

Forgot to define y. Fix w/another try/except block.

## Code & test

```
sage: def normal_line(a, f, x=x):
      f(x) = f
      df(x) = diff(f)
      try:
          m = 1/df(a)
          try:
              result = y == m*(x - a) + f(a)
          except NameError:
              var('y')
              result = y == m*(x - a) + f(a)
      except ZeroDivisionError:
          result = x == a
      return result
```



## Code & test

```
sage: def normal_line(a, f, x=x):
      f(x) = f
      df(x) = diff(f)
      try:
          m = 1/df(a)
          try:
              result = y == m*(x - a) + f(a)
          except NameError:
              var('y')
              result = y == m*(x - a) + f(a)
      except ZeroDivisionError:
          result = x == a
      return result
sage: normal_line(1, x**2)
y == 1/2*(x - 1)^2 + 1
```

# Should you use exceptions?

- Discouraged in some languages (C++), *but...*
- encouraged in Python/Sage

# Should you use exceptions?

- Discouraged in some languages (C++), *but...*
- encouraged in Python/Sage
- Helps avoid massively nested if/elif/else statements, *and...*
- more readable/understandable...
- *especially* if you name expected exception(s)

## Should you use exceptions?

- Discouraged in some languages (C++), *but...*
- encouraged in Python/Sage
- Helps avoid massively nested `if/elif/else` statements, *and...*
- more readable/understandable...
- *especially* if you name expected exception(s)
- Relatively small performance penalty

# Can raise own exceptions

If need be, you can raise exceptions from your own code:  
raise *ExceptionType*, *string\_message*

# Can raise own exceptions

If need be, you can raise exceptions from your own code:

raise *ExceptionType*, *string\_message*

Book gives example from finding determinant

# Outline

- 1 Decision-making
  - Concavity
  - Method of bisection
  - Piecewise function
- 2 Boolean statements
- 3 Having said all that...
- 4 Breaking loops
- 5 Raising and handling exceptions
- 6 Summary

# Summary

- Decision making accomplished via `if-elif-else`
  - pseudocode: **if, else if, else**
- Mathematical examples abound!
  - testing properties of functions
  - piecewise functions
  - finding roots
  - determinants of matrices
- Boolean algebra helps create conditions for `if` and `elif`
  - **and, or, not**
  - `<=`, `!=`, `>=`
- Sometimes better to catch and handle exceptions