Science & Technology Facilities Council | UK Research and Innovation | NERC SCIENCE OF THE ENVIRONMENT

Python

Control Flow

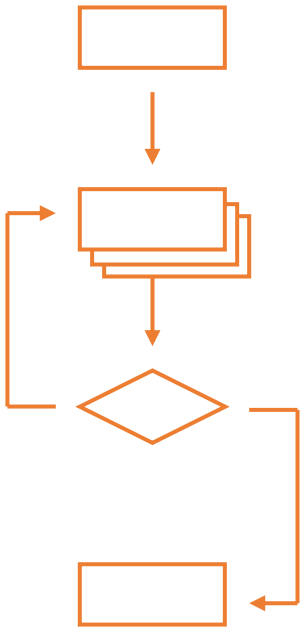
Real power of programs comes from:

Real power of programs comes from:

repetition

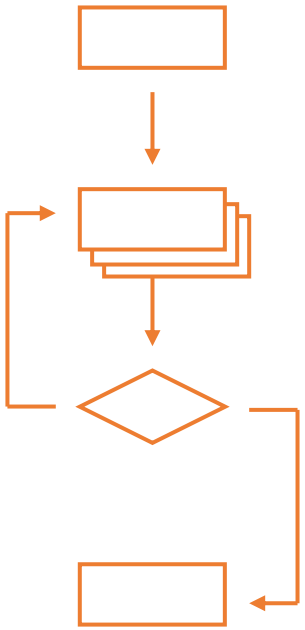
Real power of programs comes from:

repetition



Real power of programs comes from:

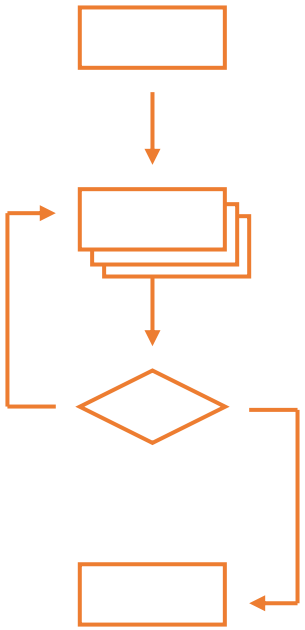
repetition



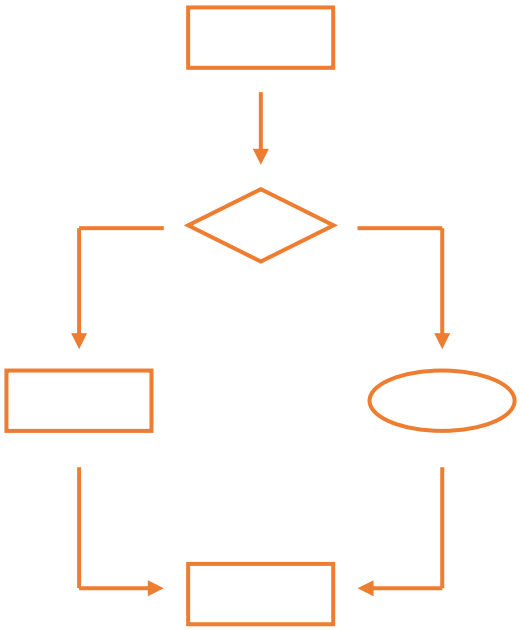
selection

Real power of programs comes from:

repetition



selection



Simplest form of repetition is *while loop*

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

test

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

do

```
3
```

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

← test again

```
3
```

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

```
3
2
```

Simplest form of repetition is *while loop*

```
num_moons = 3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
```

```
3
2
1
```

While loop may execute zero times

While loop may execute zero times

```
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

While loop may execute zero times

```
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

← not true when first tested…

While loop may execute zero times

```
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

…so this is never executed

While loop may execute zero times

```
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

```
before
after
```

While loop may execute zero times

```
print('before')
num_moons = -3
while num_moons > 0:
    print(num_moons)
    num_moons -= 1
print('after')
```

```
before
after
```

Important to consider this case when designing

and testing code

While loop may also execute forever

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
```

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
```

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
```

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
3
```

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
3
⋮
```

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
3
⋮
```

Nothing in here changes the loop control condition

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
before
3
3
3
⋮
```

Usually not the desired behavior…

While loop may also execute forever

```
print('before')
num_moons = 3
while num_moons > 0:
    print(num_moons)
print('after')
```

```
before
3
3
3
⋮
```

Usually not the desired behavior…

…but there *are* cases where it's useful

Why indentation?

Why indentation?

Studies show that's what people actually pay attention to

Why indentation?

Studies show that's what people actually pay attention to

- Every textbook on C or Java has examples where indentation and braces don't match

Why indentation?

Studies show that's what people actually pay attention to

- Every textbook on C or Java has examples where indentation and braces don't match

Doesn't matter how much you use, but whole block must be consistent

Why indentation?

Studies show that's what people actually pay attention to

- Every textbook on C or Java has examples where indentation and braces don't match

Doesn't matter how much you use, but whole block must be consistent

Python Style Guide (PEP 8) recommends 4 spaces

Why indentation?

Studies show that's what people actually pay attention to

- Every textbook on C or Java has examples where indentation and braces don't match

Doesn't matter how much you use, but whole block must be consistent

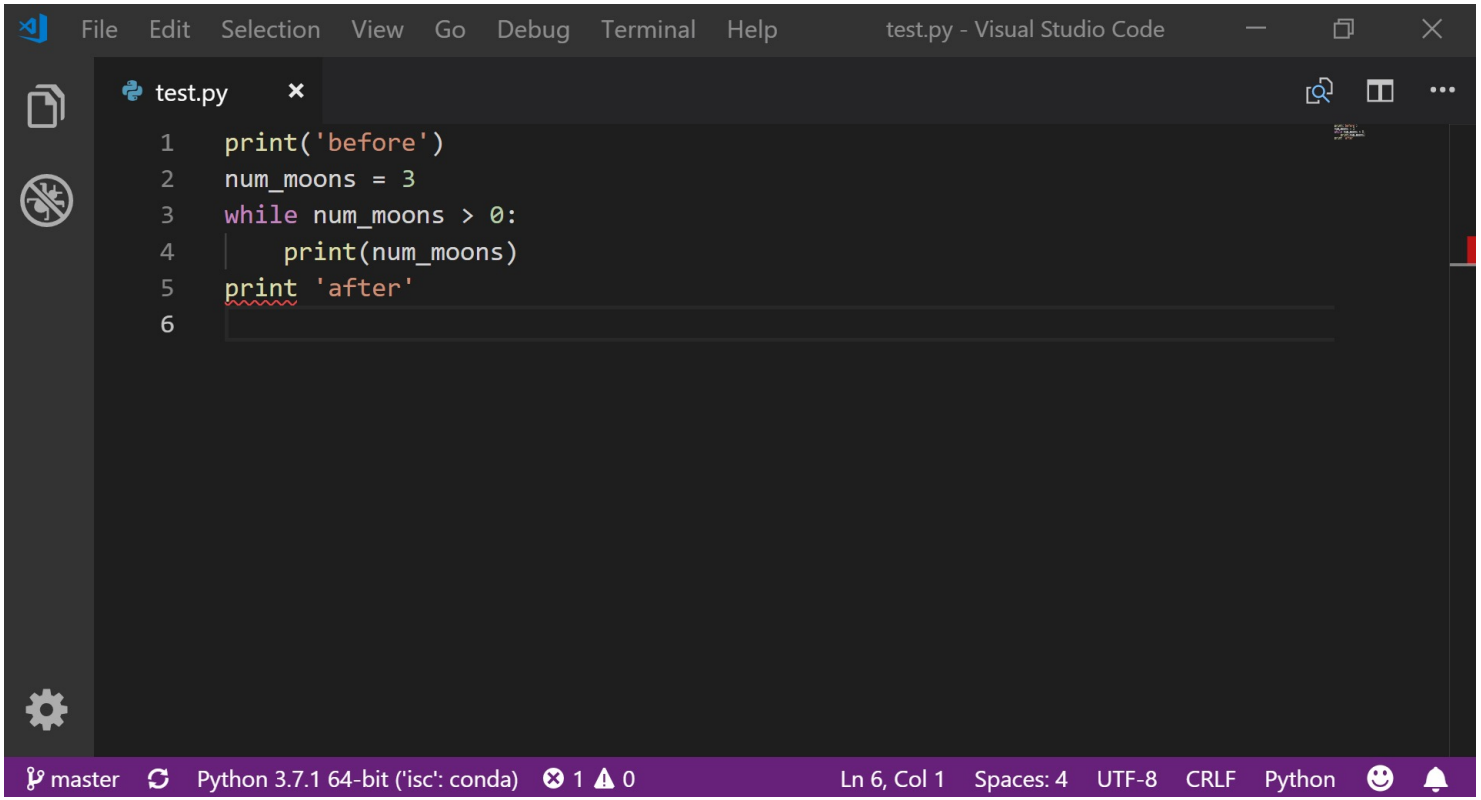
Python Style Guide (PEP 8) recommends 4 spaces

And no tab characters

Side note on IDEs (Integrated Development Environments)

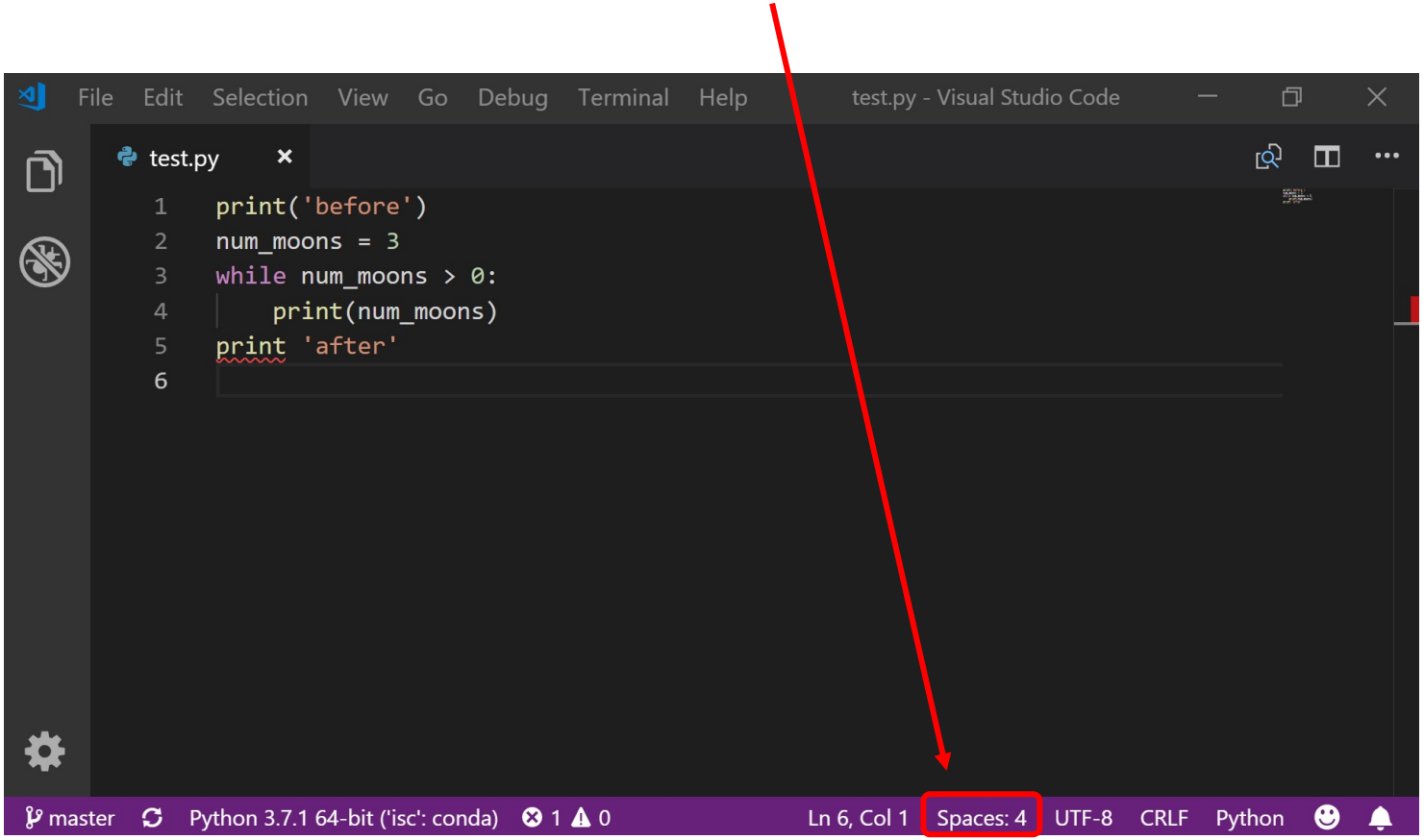
Side note on IDEs (Integrated Development Environments)

An IDE is a nicer place to write, edit and run code from all in one. Most often also include syntax highlighting, error highlighting and debugging built in (debugging will be taught later in the course).



Side note on IDEs (Integrated Development Environments)

Most IDEs will also let you choose your indentation too,
so you don't have to manually type 4 spaces…



Use `if`, `elif`, and `else` to make choices

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

← not true when first tested…

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

...so this is *not* executed

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

← this isn't true either...

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

...so this isn't executed

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

← nothing else has executed…

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```


...so this *is* executed

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

```
greater
```

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

```
greater
```

Always start with **`if`**

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

```
greater
```

Always start with **`if`**

Can have any number of **`elif`** clauses (including none)

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

```
greater
```

Always start with **`if`**

Can have any number of **`elif`** clauses (including none)

And the **`else`** clause is optional

Use `if`, `elif`, and `else` to make choices

```
moons = 3
if moons < 0:
    print('less')
elif moons == 0:
    print('equal')
else:
    print('greater')
```

```
greater
```

Always start with **`if`**

Can have any number of **`elif`** clauses (including none)

And the **`else`** clause is optional

Always tested in order

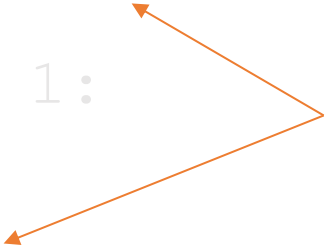
Blocks may contain blocks

Blocks may contain blocks

```
num = 0
while num <= 10:
    if (num % 2) == 1:
        print(num)
    num += 1
```

Blocks may contain blocks

```
num = 0
while num <= 10:
    if (num % 2) == 1:
        print(num)
    num += 1
```



Count from 0 to 10

Blocks may contain blocks

```
num = 0
while num <= 10:
    if (num % 2) == 1:
        print(num)
    num += 1
```

Print odd numbers

Blocks may contain blocks

```
num = 0
while num <= 10:
    if (num % 2) == 1:
        print(num)
    num += 1
```

```
1
3
5
7
9
```

A better way to do it

A better way to do it

```
num = 1
while num <= 10:
    print(num)
    num += 2
```

A better way to do it

```
num = 1
while num <= 10:
    print(num)
    num += 2
```

```
1
3
5
7
9
```

Stop here

Print primes less than 1000

Print primes less than 1000

```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```

Print primes less than 1000

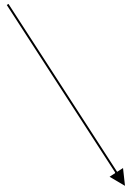
```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```

Cannot be evenly divided

by any other integer

Print primes less than 1000

```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```



```
            is_prime = True
            trial = 2
            while trial < num:
                if ...num divisible by trial...:
                    is_prime = False
                trial += 1
```

Print primes less than 1000

```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```

```
is_prime = True
trial = 2
while trial < num:
    if ...num divisible by trial...:
        is_prime = False
    trial += 1
```

Remainder is zero

Print primes less than 1000

```
num = 2
while num <= 1000:
    ...figure out if num is prime...
    if is_prime:
        print(num)
    num += 1
```

```
is_prime = True
trial = 2
while trial < num:
    if ...num divisible by trial...:
        is_prime = False
    trial += 1
```

```
(num % trial) == 0
```

Print primes less than 1000

```
num = 2
while num <= 1000:
    is_prime = True
    trial = 2
    while trial < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

A more efficient way to do it

A more efficient way to do it

```
num = 2
while num <= 1000:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

A more efficient way to do it

```
num = 2
while num <= 1000:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

N cannot be divided evenly by any number greater than sqrt(N)

Any code that hasn't been tested is probably wrong

Any code that hasn't been tested is probably wrong

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

Any code that hasn't been tested is probably wrong

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

```
2
3
4
5
7
9
```

Any code that hasn't been tested is probably wrong

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

```
2
3
4
5
7
9
```

Any code that hasn't been tested is probably wrong

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

```
2
3
4
5
7
9
```

Where's the bug?

Failures occur for perfect squares

Failures occur for perfect squares

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

Failures occur for perfect squares

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

2**2 == 4

Failures occur for perfect squares

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

2**2 == 4

So never check to see

if 4 % 2 == 0

Failures occur for perfect squares

```
num = 2
while num <= 10:
    is_prime = True
    trial = 2
    while trial**2 < num:
        if (num % trial) == 0:
            is_prime = False
        trial += 1
    if is_prime:
        print(num)
    num += 1
```

2**2 == 4

So never check to see

if 4 % 2 == 0

Or if 9 % 3 == 0, etc.

More ways to control flow while inside a loop:

`break`, `continue`, `pass`

More ways to control flow while inside a loop:

`break`, `continue`, `pass`

e.g. Print the first multiple of a given value

`break`, `continue`, `pass`

e.g. Print the first multiple of a given value

```
value = 14
trial = 2
while trial < value:
    if trial % value == 0:
        print(trial)
        break
    trial += 1
```

```
2
```

break, continue, pass

e.g. Print the first odd multiple of a given value

```
value = 14
trial = 2
while trial < value:
    if trial % 2 == 0:
        trial += 1
        continue
    if trial % value == 0:
        print(trial)
        break
    trial += 1
```

```
7
```

`break`, `continue`, `pass`

If you get to a point in your logic where you want to specifically do nothing, you can use `pass`

```
value = 14
trial = 2
while trial < value:
    if trial % 2 == 0:
        pass
    if trial % value == 0:
        print(trial)
        break
    trial += 1
```

2

Python

Common operators: `and`, `not` and `or`

Testing expressions

You may want to create a more complex expression when testing using `if` or `while`.

Testing expressions

You may want to create a more complex expression when testing using `if` or `while`.

```
age = 23
name = "Jemma"
height = 1.63
```

Some variables

Testing expressions

You may want to create a more complex expression when testing using `if` or `while`.

```
age = 23
name = "Jemma"
height = 1.63
```

Some variables

What if we want someone with all of these qualities?
What if we want someone who is some of these qualities?

Introducing the `and`, `not` and `or` operators

You may want to create a more complex expression when testing using `if` or `while`.

```
age = 23
name = "Jemma"
height = 1.63
if age == 23:
    print("Correct age")
if name == "Jemma":
    print("Correct name")
if height == 1.63:
    print("Correct height")
```

Introducing the `and`, `not` and `or` operators

You may want to create a more complex expression when testing using `if` or `while`.

```
age = 23
name = "Jemma"
height = 1.63
if age == 23:
    print("Correct age")
if name == "Jemma":
    print("Correct name")
if height == 1.63:
    print("Correct height")
```

Could perform 3 tests to make sure the 3 variables are correct

This seems inefficient

Using `and`

Block executes only if both expressions return `True`

```
age = 23
name = "Jemma"
height = 1.63
if name == "Jemma" and age == 23:
    print("It is Jemma!")
```

```
It is Jemma!
```

Using `and`

`and` can be chained more than once too:

```
age = 23
name = "Jemma"
height = 1.63
if name == "Jemma" and age >= 20 and height < 2:
    print("It is like Jemma!")
```

```
It is like Jemma!
```

Using `not`

`not` will reverse the Boolean result of an expression, we can use it to make blocks that execute only if the expression returns `False` or `None`

```
age = 22
name = "Rachel"
height = 1.65
if not name == "Jemma":
    print("It isn't Jemma!")
```

```
It isn't Jemma!
```

Using `not`

Can be used to see if variable not in a collection:

```
x = 25
if x not in [1, 2, 3]:
    print("Didn't find x in list")
```

```
Didn't find x in list
```

Can even write `is not` to compare:

```
if x is not 100:
    print("x is not 100")
```

```
x is not 100
```

Using `or`

`or` will return `True` if either or both expressions are `True`:

```
greeting = "Hello"
if greeting == "Hi" or greeting == "Hello":
    print("Good day")
```

```
Good day
```

Can be chained more than once:

```
x = 25
if x < 0 or x > 100 or x == 25:
    print("x is correct")
```

```
x is correct
```

Chaining all of these operators

All of these operators can be chained together to create more complex expressions:

```
start = False
end = 55
status = "STARTED"
if status == "STARTED" and (start is not False or end > 0):
    print("Running")
```

Running

You might need brackets (as above) to specify the precedence of evaluation of expressions.

created by

Greg Wilson

September 2010