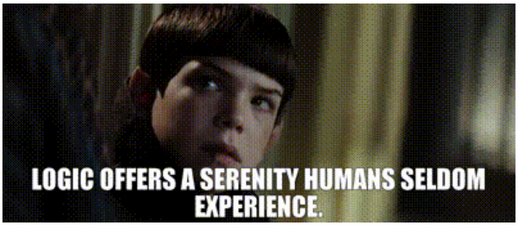
Last time:

Logic



- propositions
- operators
- expressions

Truth table homework

De Morgan's Laws

Expression negation

$$\neg\left(P \wedge Q\right) = \neg P \vee \neg Q$$

$$\neg\left(P \vee Q\right) = \neg P \wedge \neg Q$$

More complex example:

$$\neg\left(P \wedge \neg Q \vee R\right) = \neg\left(\left(P \wedge \neg Q\right) \vee R\right)$$
$$= \neg(P \wedge \neg Q) \wedge \neg R = (\neg P \vee \neg\neg Q) \wedge \neg R$$
$$= (\neg P \vee Q) \wedge \neg R$$

It's often the case that we have a complex logical expression and we need to find its opposite.

Say that it's safe to land an aircraft if the landing lights are on, the gear is down and ATC has given permission to land. What's the opposite of that logical expression? If ____________ of those conditions are not met, ___________________________.

Previously:

Expressions

Values and *operations* that *evaluate* to a *value*

Values

Literals

- integer literals (e.g., `42`, `1_000_000`)
- *floating-point* (real-valued) literals (e.g., `3.14`, `1e6`)
- *imaginary* literals (e.g., `3.14j`)
- *string* literals (e.g., `'hello'`)
- *Boolean* (logical) literals (e.g., `True`)

Operations

Arithmetic operators ✅

Function calls ✅

Logical operators

Comparators

Logical operators

Boolean operands *and* result

Operator	Kind	Evaluates to true iff:
`and`	Binary	both operands are true
`or`	Binary	at least one operand is true
`!=` or `^`	Binary	exactly one operand is true
`not`	Unary	the operand is false

Comparators

Evaluate to `True` or `False`

Logical or *Boolean* values

Operator	Operation
`<`	Less than?
`>`	Greater than?
`<=`	Less than or equal to?
`>=`	Greater than or equal to?
`==`	Equal to?
`!=`	Not equal to?

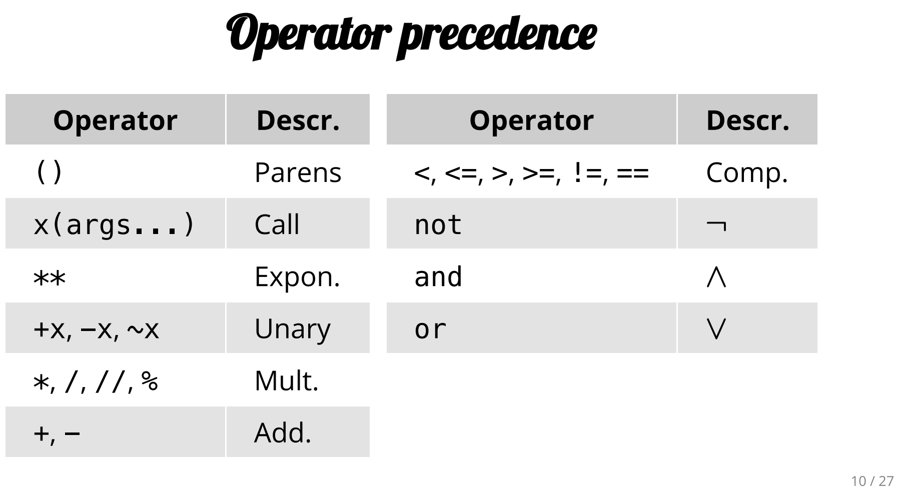
Operations

Arithmetic operators ✅

Function calls ✅ (for now)

Logical operators ✅

Comparators ✅



Operator precedence

Operator	Descr.	Operator	Descr.
()	Parens	<, <=, >, >=, !=, ==	Comp.
x(args...)	Call	not	¬
**	Expon.	and	∧
+x, −x, ~x	Unary	or	∨
*, /, //, %	Mult.		
+, −	Add.		

A more complete order of operations (including operators that we will only get to later in the course) can be found here:

Operator	Description
()	Parentheses
x[i], x[i:j], x(args...), x.attribute	Subscription, slicing, call, attribute reference
**	Exponentiation
+x, −x, ~x	Unary positive, negative, bitwise not
*, @, /, //, %	Multiplication
+, −	Addition and subtraction
<<, >>	Shifts
&	Bitwise and
^	Bitwise xor
|	Bitwise or
in, not in, <, <=, >, >=, !=, ==	Comparisons and membership tests
not	Boolean not
and	Boolean and
or	Boolean or
if − else	Conditional expression
=	Assignment

The absolute full details can be seen at https://docs.python.org/3/reference/expressions.html#operator-precedence, but that includes some operators that we won't even get to in this course.

and now:

Variables!

… in mathematics and in programming, being in some ways similar and in others different

Variables

We previously *used* variables:

```
>>> from math import *
>>> pi
3.141592653589793
>>> 2j * pi
6.283185307179586j
```

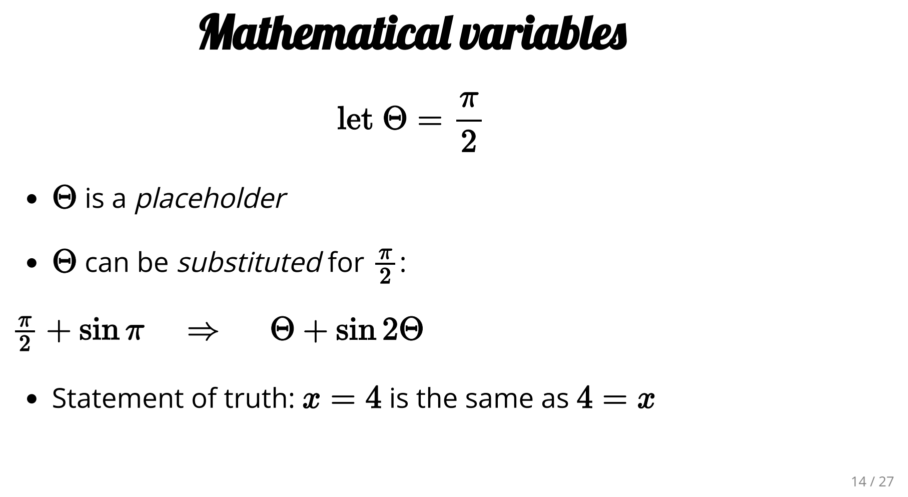
```
angle*6*E*I/(x*(3*L**2-3*L*x+x**2))

(angle*(6*E*I))/((3*L**2-3*L*x+x**2)*x)

(6*E*I*angle)/(x*((3*(L**2))-(3*L*x)+(x**2)))

(angle*6*E*I)/(3*L*L*x)-(3*L*x*x)+(x*x*x)
```

Making variables

But where do variables *come* from?

First: what *are* variables?

- in mathematics
- in programming



Mathematical variables

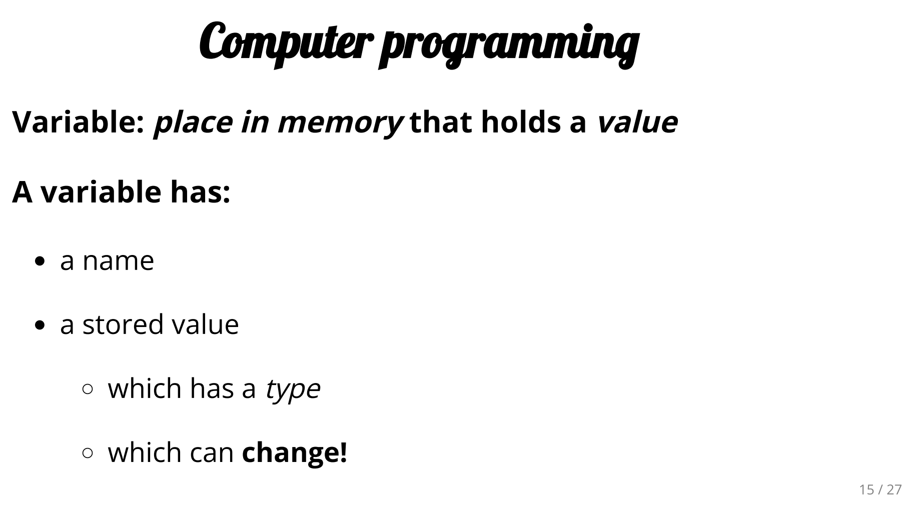
$$\text{let } \Theta = \frac{\pi}{2}$$

- Θ is a *placeholder*
- Θ can be *substituted* for $\frac{\pi}{2}$:

$$\frac{\pi}{2} + \sin \pi \quad \Rightarrow \quad \Theta + \sin 2\Theta$$

- Statement of truth: $x = 4$ is the same as $4 = x$

You should already have some familiarity with the concept of variables from mathematics. In math, we describe variables using "let" statements, e.g., $\text{let } x = \frac{2\pi}{3}$. In this usage, ____________________ ____________________________, i.e., wherever you see the variable x you can substitute in the value $\frac{2\pi}{3}$ instead. If you wanted to express a changing value of x, you might use names like x' or x'', which are clearly related but are in fact __________________________: x is not the same variable as x'.



Computer programming

Variable: *place in memory* that holds a *value*

A variable has:

- a name
- a stored value
  - which has a *type*
  - which can **change!**

In computer programming, variables are a related but ____________________ concept. Variables in programming languages, like in mathematics are names that can be used to refer to values. Unlike mathematics, however, programming variables do not refer directly to values but rather than __________________________________. For example, the image to the right depicts three integer values (42, 17 and 54) being held at three different locations in memory. It is worth noting a couple of things about these variables:

1. we refer to each one by a **name**,
2. we talk about each one in terms of a **type** (integer vs real vs ...),
3. each has a specific **size** in memory (which depends on the type) and
4. each has a defined **place** in memory.

Show the `type()` function

Creating variables

First (real) use of Python *statements**

```
n = 42
```

This defines:

- the *name* of the variable (n)
- the initial *value* stored in the variable

A Python file (or "script") is a series of statements that can be run one after the other.

Initial value?

Mathematically invalid:

$$\text{let } n = 1$$

$$\text{let } n = 2$$

Programming variables can change value:

```
n = 1
n = 2
```

Exercises

1. In a Python interpreter, define four variables:
   - one containing an *integer*
   - one containing a *floating-point* number
   - one containing a *string*
   - one containing a *Boolean* value
2. Change their values; check their types with `type()`

Variables and types

Strictly speaking:

- **variables** don't have types, **values** do
- when someone says, "the type of a variable", their more precise meaning is... the *type of its current value*

Types... so far...

Give two examples for each of:

Type	Used for
`bool`	Boolean (true/false) values
`int`	"Whole" things
`float`	Real numbers
`complex`	Complex numbers
`str`	Names, arbitrary-length text

Type conversions

We've see `type()` (which does what, again?)

How about converting values to *different* types?

Why would we want to do this?
How would we do this?

Demonstrate:

- `type()`
- `int()`
- `round()`
- `float()`
- `str()`

Example

```
name = input('What is your name? ')
quest = input('What is your quest? ')
v = input('What is the average airspeed velocity of an unladen swallow? ')
```

How long will it take the swallow to fly 792 m?

Type conversions are very useful!

Show error from `792 / v` without type conversion

Monotony and tedium

Speaking of useful...

- re-writing the same statements over and over is tedious
- programming's supposed to make life *less* tedious!
- enter the *script* mode of programming

A Python script is named this way because

______________ how to create and run a script with IDLE and Thonny.

Python scripts

```python
# Copyright (c) 2020 Jonathan Anderson
# Permission is granted to copy and modify this code for any purpose.
#
# This is an example of a simple Python script for questing.

# First, gather some basic information from the user.
name = input('What is your name? ')
quest = input('What is your quest? ')
v = input('What is the average airspeed of an unladen swallow? ')

# Greet the brave dame or knight.
print(f'Greetings, brave {name}!')

# Convert speed into a floating-point number and calculate quest time.
v = float(v)
print(f'Your quest for {quest} will be aided when the swallow flies 792 m.')
print(f'This should take {792 / v} s.')
```

What can we see in this file (which is available as python.py)?

- comments
- copyright
- type conversion
- variables

What we just saw

Comments

```
# Helpful descriptive text for people, not the computer!
# Everything from # to the end of the line
```

Copyright

- writing code is a *creative act*
- more like an English assignment than Math!

What's next?

Write a Python script!

- take some input, compute something, produce output
- good preparation for exercise 2