Surprisingly Unsurprising

The joy of unexpected simplicity

Matthew Stephen Stuckwisch

Surprisingly Unsurprising

The joy of unexpected simplicity

and how to bring it to end users

Matthew Stephen Stuckwisch

Sosiegu

(Calm)

Marta Mori d’Arriba

Préstame la sorpresa,
préstame aquello
colo que nun cuntaba:
el soníu del agua
no fondero la viesca,
los finales abiertos
y ciertes charres nocturnes
que, como’l cursu d’un ríu,
sábese y nun se sabe
el sitiu onde nos pueden amenar

Préstame la sorpresa,
préstame lo fortuito y lo casual.
Por eso ye que naguo pol sosiegu
que quiciabes acabe traeme
imprevisiblemente al prósimu momentu.

Sosiegu

(Calm)

Marta Mori d’Arriba

Préstame la sorpresa,	*I like surprises,*
préstame aquello	*I like those things*
colo que nun cuntaba:	*I can’t anticipate:*
el soníu del agua	*the sound of water*
no fondero la viesca,	*in the depths of the forest,*
los finales abiertos	*open endings*
y ciertes charres nocturnes	*and those late night chats*
que, como’l cursu d’un ríu,	*that, like the flow of a river,*
sábese y nun se sabe	*can lead us to places*
el sitiu onde nos pueden amenar	*known and unknown.*
Préstame la sorpresa,	*I like surprises,*
préstame lo fortuito y lo casual.	*I like the chance and fortune.*
Por eso ye que naguo pol sosiegu	*For this I yearn for the calm*
que quiciabes acabe traeme	*that might end up bringing me*
imprevisiblemente al prósimu momentu.	*chancefully to the next moment.*

Goals of this talk

1. Describe surprising(ly mundane) features of Raku.
2. Consider how they might be used in module design.
3. Demo some ways to (re)create some potentially useful things.
4. Show ways existing modules have approached things to stay Raku-ish.
5. Provide a rough checklist for module development.

```
0.1 + 0.2 = ____
```

0.1 + 0.2 = ____

a) 0.30000000000000004

b) 0.3

`0.1 + 0.2 = ____`

`a) 0.30000000000000004`

C, Java, JavaScript,
Julia, Python 2*/3,
Perl*, Ruby, Rust,
Swift

`b) 0.3`

`0.1 + 0.2 = ____`

`a) 0.30000000000000004`

C, Java, JavaScript, Julia, Python 2*/3, Perl*, Ruby, Rust, Swift

`b) 0.3`

SageMath, R, Mathematica, MATLAB

`0.1 + 0.2 = ____`

`a) 0.30000000000000004`

C, Java, JavaScript, Julia, Python 2*/3, Perl*, Ruby, Rust, Swift

`b) 0.3`

SageMath, R, Mathematica, MATLAB

* These languages cheat and stringify by default as 0.3 because of trimming, but internally they store/use the wrong value.

`0.1 + 0.2 = ____`

`a) 0.30000000000000004`

C, Java, JavaScript, Julia, Python 2*/3, Perl*, Ruby, Rust, Swift

`b) 0.3`

SageMath, R, Mathematica, MATLAB Raku

* These languages cheat and stringify by default as 0.3 because of trimming, but internally they store/use the wrong value.

In Raku, the two most basic class types (numbers, strings) are chosen smartly:

In Raku, the two most basic class types
(numbers, strings) are chosen smartly:

Numbers prefer rational / big integer types;
strings default to a grapheme-based Unicode.

In Raku, the two most basic class types (numbers, strings) are chosen smartly:

Numbers prefer rational / big integer types; strings default to a grapheme-based Unicode.

Smart defaults save people time that they might not even know they're otherwise losing.

Switching

Switching

Raku doesn’t use a traditional switch statement. Instead, it uses **`given`**

Switching

Raku doesn’t use a traditional switch statement. Instead, it uses **`given`**

```
given $foo {
    when    1 { … }
    when    2 { … }
    when    3 { … }
    default   { … }
}
```

Switching

```
given $foo {
    when    'a' { … }
    when     1  { … }
    when    /α/ { … }
    default     { … }
}
```

Switching

```
given $foo {
    when     'a' { … }
    when      1  { … }
    when     /α/ { … }
    default      { … }
}
```

Switching

```
given $foo, $bar {
    when     'a', 'b' { … }
    when      1,   2  { … }
    when     /α/, /β/ { … }
    default           { … }
}
```

Switching

```
given $foo, $bar {
    when    'a', 'b' { … }
    when     1,   2  { … }
    when    /α/, /β/ { … }
    when     *,  Str { … }
    default          { … }
}
```

Switching

```
if    ($foo, $bar) ~~ ('a', 'b') { … }
elsif ($foo, $bar) ~~ ( 1,   2 ) { … }
elsif ($foo, $bar) ~~ (/α/, /β/) { … }
elsif ($foo, $bar) ~~ ( *,  Str) { … }
else                             { … }
```

Switching

```
if    ($foo, $bar) ~~ ('a', 'b') { … }
elsif ($foo, $bar) ~~ ( 1,   2 ) { … }
elsif ($foo, $bar) ~~ (/α/, /β/) { … }
elsif ($foo, $bar) ~~ ( *,  Str) { … }
else                             { … }
```

Switching

```
if $foo ~~ 'a'
&& $bar ~~ 'b'
```

```
if    ($foo, $bar) ~~ ('a', 'b') { … }
elsif ($foo, $bar) ~~ ( 1,   2 ) { … }
elsif ($foo, $bar) ~~ (/α/, /β/) { … }
elsif ($foo, $bar) ~~ ( *,  Str) { … }
else                             { … }
```

Switching

```
if $foo ~~ 'a'
&& $bar ~~ 'b'
```

```
if    ($foo, $bar) ~~ ('a', 'b') { … }
elsif ($foo, $bar) ~~ ( 1,   2 ) { … }
elsif ($foo, $bar) ~~ (/α/, /β/) { … }
elsif ($foo, $bar) ~~ ( *,   Str) { … }
else                             { … }
```

Switching

```
if $foo ~~ 'a'
&& $bar ~~ 'b'
```

```
if    ($foo, $bar) ~~ ('a', 'b') { … }
elsif ($foo, $bar) ~~ ( 1,   2 ) { … }
elsif ($foo, $bar) ~~ (/α/, /β/) { … }
elsif ($foo, $bar) ~~ ( *,  Str) { … }
else                             { … }
```

* (whatever) means "I don't care about this value", it always returns True!

Switching

```
if $foo ~~ 'a'
&& $bar ~~ 'b'
```

```
if    ($foo, $bar) ~~ ('a', 'b') { … }
elsif ($foo, $bar) ~~ ( 1,   2 ) { … }
elsif ($foo, $bar) ~~ (/α/, /β/) { … }
elsif ($foo, $bar) ~~ ( *,   Str) { … }
else                             { … }
```

* (whatever) means "I don't care about this value", it always returns True!

Str typechecks for Str

Switching

```
if $foo ~~ 'a'
&& $bar ~~ 'b'
```

```
if    ($foo, $bar) ~~ ('a', 'b') { … }
elsif ($foo, $bar) ~~ ( 1,   2 ) { … }
elsif ($foo, $bar) ~~ (/α/, /β/) { … }
elsif ($foo, $bar) ~~ ( *,   Str) { … }
else                             { … }
```

* (whatever) means "I don't care about this value", it always returns True!

Str typechecks for Str

```
given $foo, $bar {
    when    'a', 'b' { … }
    when     1,   2  { … }
    when    /α/, /β/ { … }
    when     *,  Str { … }
    default          { … }
}
```

Junctions

Junctions

You never know what you have until it’s gone.

Junctions

You never know what you have until it’s gone.

```
my @a = <a b c d e f g h>;
```

Junctions

You never know what you have until it’s gone.

```
my @a = <a b c d e f g h>;
my @b = <i j k l m n o b>;
```

Junctions

You never know what you have until it’s gone.

```
my @a = <a b c d e f g h>;
my @b = <i j k l m n o b>;
my @c = <a b b c c d e e>;
```

Junctions

You never know what you have until it’s gone.

```
my @a = <a b c d e f g h>;
my @b = <i j k l m n o b>;
my @c = <a b b c c d e e>;
```

Junctions

You never know what you have until it's gone.

```
my @a = <a b c d e f g h>;
my @b = <i j k l m n o b>;
my @c = <a b b c c d e e>;

say "overlap"   if any @a eq any @b;
```

Junctions

You never know what you have until it’s gone.

```
my @a = <a b c d e f g h>;
my @b = <i j k l m n o b>;
my @c = <a b b c c d e e>;

say "overlap"   if any @a eq any @b;
say "all-valid" if all @c eq any @a;
```

Junctions

You never know what you have until it’s gone.

```
my @a = <a b c d e f g h>;
my @b = <i j k l m n o b>;
my @c = <a b b c c d e e>;

say "overlap"   if any @a eq any @b;
say "all-valid" if all @c eq any @a;
```

Junctions

You never know what you have until it’s gone.

```
my @a = <a b c d e f g h>;
my @b = <i j k l m n o b>;
my @c = <a b b c c d e e>;

say "overlap"   if any @a eq any @b;
say "all-valid" if all @c eq any @a;
```

```
'overlap'
```

Junctions

You never know what you have until it’s gone.

```
my @a = <a b c d e f g h>;
my @b = <i j k l m n o b>;
my @c = <a b b c c d e e>;

say "overlap"   if any @a eq any @b;
say "all-valid" if all @c eq any @a;
```

```
'overlap'
'all-valid'
```

The slurpy family

The slurpy family

Three ways to consume lists of items.

The slurpy family

Three ways to consume lists of items.

```
sub slurpy ( *@pour-and-savor ) { … }
```

The slurpy family

Three ways to consume lists of items.

```
sub slurpy ( *@pour-and-savor ) { … }
sub slurpy (**@chug-no-regrets) { … }
```

The slurpy family

Three ways to consume lists of items.

```
sub slurpy ( *@pour-and-savor ) { … }
sub slurpy (**@chug-no-regrets) { … }
sub slurpy ( +@read-the-label ) { … }
```

The slurpy family

The slurpy family

```
@*pour-and-savor
```

The slurpy family

`@*pour-and-savor`	Items inside of lists are iterated

The slurpy family

`@*pour-and-savor` Items inside of lists are iterated

```
sub parrot (*@x) { .say for @x }
parrot 1, (2, 3, (4, 5), 6), 7, (((8),),)
my @abc = <a b c>; parrot @abc;
```

The slurpy family

`@*pour-and-savor` Items inside of lists are iterated

```
sub parrot (*@x) { .say for @x }
parrot 1, (2, 3, (4, 5), 6), 7, (((8),),)
my @abc = <a b c>; parrot @abc;
```

```
a
b
c
```

The slurpy family

`@*pour-and-savor` Items inside of lists are iterated

```
sub parrot (*@x) { .say for @x }
parrot 1, (2, 3, (4, 5), 6), 7, (((8),),)
my @abc = <a b c>; parrot @abc;
```

```
1     a
2     b
3     c
4
5
6
7
```

The slurpy family

`@**chug-no-regrets` A list treated as its

```
sub parrot (**@x) { .say for @x }
parrot 1, (2, 3, (4, 5), 6), 7, (((8),),)
my @abc = <a b c>; parrot @abc;
```

The slurpy family

`@**chug-no-regrets` A list treated as its

```
sub parrot (**@x) { .say for @x }
parrot 1, (2, 3, (4, 5), 6), 7, (((8),),)
my @abc = <a b c>; parrot @abc;
```

```
(a b c)
```

The slurpy family

`@**chug-no-regrets` A list treated as its

```
sub parrot (**@x) { .say for @x }
parrot 1, (2, 3, (4, 5), 6), 7, (((8),),)
my @abc = <a b c>; parrot @abc;
```

```
1                (a b c)
(2 3 (4 5) 6)
7
((8))
```

The slurpy family

`@+read-the-label` Decide smartly (by single argument rule)

```
sub parrot (+@x) { .say for @x }
parrot 1, (2, 3, (4, 5), 6), 7, (((8),),)
my @abc = <a b c>; parrot @abc;
```

The slurpy family

`@+read-the-label` Decide smartly (by single argument rule)

```
sub parrot (+@x) { .say for @x }
parrot 1, (2, 3, (4, 5), 6), 7, (((8),),)
my @abc = <a b c>; parrot @abc;
```

```
a
b
c
```

The slurpy family

`@+read-the-label` Decide smartly (by single argument rule)

```
sub parrot (+@x) { .say for @x }
parrot 1, (2, 3, (4, 5), 6), 7, (((8),),)
my @abc = <a b c>; parrot @abc;
```

```
1                    a
(2 3 (4 5) 6)        b
7                    c
((8))
```

The slurpy family

```
sub slurpy ( *@pour-and-savor ) { … }
sub slurpy (**@chug-no-regrets) { … }
sub slurpy ( +@read-the-label ) { … }
```

The slurpy family

```
sub slurpy ( *@pour-and-savor ) { … }
sub slurpy (**@chug-no-regrets) { … }
sub slurpy ( +@read-the-label ) { … }
```

```
say @a, $b, $c;          say @a

for @a, $b, $c {…}       for @a {…}
```

Parentheses

Parentheses

In Raku, parentheses don't make a list. [pikachu_face.gif]

Parentheses

In Raku, parentheses don't make a list. [pikachu_face.gif]

Parentheses are more likely to be superfluous.

Parentheses

In Raku, parentheses don't make a list. [pikachu_face.gif]

Parentheses are more likely to be superfluous.

```
my @foo = 1, 2, 3;
```

Parentheses

In Raku, parentheses don't make a list. [pikachu_face.gif]

Parentheses are more likely to be superfluous.

```
my @foo = 1, 2, 3;
```

Even for sub/method calls

Parentheses

In Raku, parentheses don't make a list. [pikachu_face.gif]

Parentheses are more likely to be superfluous.

```
my @foo = 1, 2, 3;
```

Even for sub/method calls

```
bar($foo, $a, $b)
bar $foo, $a, $b
```

Parentheses

In Raku, parentheses don't make a list. [pikachu_face.gif]

Parentheses are more likely to be superfluous.

```
my @foo = 1, 2, 3;
```

Even for sub/method calls

```
bar($foo, $a, $b)
bar $foo, $a, $b
```

```
                 $foo.bar( $a, $b)
OO style         $foo.bar: $a, $b
Procedural style bar $foo: $a, $b
```

Parentheses

These can be chained too, as long as each call is the final one of the previous:

Parentheses

These can be chained too, as long as each call is the final one of the previous:

```
a(b(c(d(e(f(1,2,3))))))
```

Parentheses

These can be chained too, as long as each call is the final one of the previous:

```
a(b(c(d(e(f(1,2,3))))))
  a b c d e f 1, 2, 3
```

Parentheses

These can be chained too, as long as each call is the final one of the previous:

```
a(b(c(d(e(f(1,2,3))))))
  a b c d e f 1, 2, 3
```

```
eat bake sear butcher get $cow
```

Parentheses

These can be chained too, as long as each call is the final one of the previous:

```
a(b(c(d(e(f(1,2,3))))))
  a b c d e f 1, 2, 3
```

```
eat bake sear butcher get $cow
eat(bake(sear(butcher(get($cow)))))
```

Parentheses

These can be chained too, as long as each call is the final one of the previous:

```
a(b(c(d(e(f(1,2,3))))))
  a b c d e f 1, 2, 3

eat bake sear butcher get $cow
eat(bake(sear(butcher(get($cow)))))

say substr
      $string,
      0,
      max $string.elems, 8
```

Parentheses

Not required after control statements

Parentheses

Not required after control statements

```
if $condition { … }
```

Parentheses

Not required after control statements

```
if $condition { … }

for @list { … }
```

Parentheses

Not required after control statements

```
if $condition { … }

for @list { … }

unless $foo && $bar
    || $abc && $xyz
    || $override
{ initial-setup }
```

Parentheses

Why is this important?

Parentheses

Why is this important?

Cleaner code! Less line noise!
No parentheses hell! I love you Lisp, I promise.

Parentheses

Why is this important?

Cleaner code! Less line noise!
No parentheses hell! I love you Lisp, I promise.

On the other hand…

Parentheses

Why is this important?

Cleaner code! Less line noise!
No parentheses hell! I love you Lisp, I promise.

On the other hand…

Methods, subs and control statements can be visually similar.

Blocks

In Raku, all blocks are objects.

```
sub foo ($a) { say $a }
sub bar ($a) {  $a()  }

                           -> ;; $_ is raw = OUTER::<$_>
foo { say “surprise!” }  { #`(Block|140425853909408) … }
bar { say “surprise!” }  surprise!
```

Does that mean something like…

Does that mean something like…

```
loop { … }
```

Does that mean something like…

```
loop { … }
```

is really just a sub?

Does that mean something like…

```
loop { … }
```

is really just a sub?

Basically, yes.*

* Internally it's a bit more complicated since loop is defined in NQP and we need to handle things like last, etc., but then again everything is really just ultimately defined there as a sub or method anyways. Just shhh…

Let’s make our own loop ‘control statement’

Let’s make our own loop ‘control statement’

```
sub bucle (&código) { código( ) xx ∞ }
```

Spanish for “loop”

Spanish for “code”

Let’s make our own loop ‘control statement’

```
sub bucle (&código) { código( ) xx ∞ }
```

Spanish for “loop”

Spanish for “code”

```
bucle { say "¡Hola!" }
```

Let’s make our own loop ‘control statement’

```
sub bucle (&código) { código( ) xx ∞ }
```

Spanish for “loop”

Spanish for “code”

```
bucle { say "¡Hola!" }
```

```
¡Hola!
¡Hola!
¡Hola!
¡Hola!
¡Hola!
```

Gather / Take

Collect / Grab

Collect / Grab

```
sub collect (&code) {
  my @*collection;
  code();
  @*collection;
}
```

Collect / Grab

```
sub collect (&code) {
  my @*collection;
  code();
  @*collection;
}

sub grab ($item) {
  @*collection.push: $item;
}
```

Collect / Grab

```
sub collect (&code) {
  my @*collection;
  code();
  @*collection;
}

sub grab ($item) {
  @*collection.push: $item;
}
```

Collect / Grab

```
sub collect (&code) {
  my @*collection;
  code();
  @*collection;
}

sub grab ($item) {
  @*collection.push: $item;
}
```

```
my @primes = collect {
  grab $_
    if .is-prime
      for ^100
}

say @primes;
```

Collect / Grab

```
sub collect (&code) {
  my @*collection;
  code();
  @*collection;
}

sub grab ($item) {
  @*collection.push: $item;
}
```

```
my @primes = collect {
    grab $_
        if .is-prime
            for ^100
}

say @primes;
```

```
[2 3 5 7 11 13 17 19 23 29 31 37 41
43 47 53 59 61 67 71 73 79 83 89 97]
```

Collect / Grab

```
sub collect (&code) {
  my @*collection;
  code();
  @*collection;
}

sub grab ($item) {
  @*collection.push: $item;
}
```

```
my @six-factors =
  collect {
    grab $_ if $_ %% 2
      for collect {
        grab $_ if $_ %% 3
          for ^100
      }
  }

say @six-factors;
```

Collect / Grab

```
sub collect (&code) {
  my @*collection;
  code();
  @*collection;
}

sub grab ($item) {
  @*collection.push: $item;
}
```

```
my @six-factors =
  collect {
    grab $_ if $_ %% 2
      for collect {
        grab $_ if $_ %% 3
          for ^100
      }
  }

say @six-factors;
```

```
[0 6 12 18 24 30 36 42 48
 54 60 66 72 78 84 90 96]
```

Localized Block

Localized Block

What do we want?

Localized Block

What do we want?

```
say "Hello"; # normal say

localized {
   say "Hello"; # localized say
}

say "Good-bye"; # normal say
```

Localized Block

```
say "Hello";

localized {
   say "Hello";
}

say "Good-bye";
```

Localized Block

```
say "Hello";

localized {
   say translate "Hello";
}

say "Good-bye";
```

Localized Block

```
say "Hello";

localized {
   say "Hello";
}

say "Good-bye";
```

Localized Block

```
foo "Hello";

localized {
   foo "Hello";
}

foo "Good-bye";
```

Localized Block

Localized Block

```
sub foo($s) {
  if   ?? { say translate $s }
  else    { say           $s }
}
```

Localized Block

```
foo "Hello";

localized {
    my $*LOCALIZED = True;
    foo "Hello";
}

foo "Good-bye";
```

Localized Block

Localized Block

```
sub foo($s) {
  if   $*LOCALIZED { say translate $s }
  else             { say           $s }
}
```

Localized Block

```
foo "Hello";

localized {
   my $*LOCALIZED = True;
   foo "Hello";
}

foo "Good-bye";
```

Localized Block

Localized Block

```
sub foo($s) {
  if   $*LOCALIZED { say translate $s }
  else             { say           $s }
}
```

Localized Block

```
  sub foo($s) {
    if   $*LOCALIZED { say translate $s }
    else             { say           $s }
  }

&say.wrap: sub ($s) {
  if   $*LOCALIZED { callwith translate $s }
  else             { callsame             }
}
```

Localized Block

```
sub foo($s) {
  if   $*LOCALIZED { say translate $s }
  else             { say           $s }
}

&say.wrap: sub ($s) {
  if   $*LOCALIZED { callwith translate $s }
  else             { callsame              }
}
```

By wrapping, we don't need to call a special sub.
Wrapping is global, so the conditional ensures other calls to `say` are unchanged.

Localized Block

```
foo "Hello";

localized {
   my $*LOCALIZED = True;
   foo "Hello";
}

foo "Good-bye";
```

Localized Block

```
foo "Hello";

localized {
   my $*LOCALIZED = True;
   my $*LANGUAGE  = 'en';
    foo "Hello";
}

foo "Good-bye";
```

Localized Block

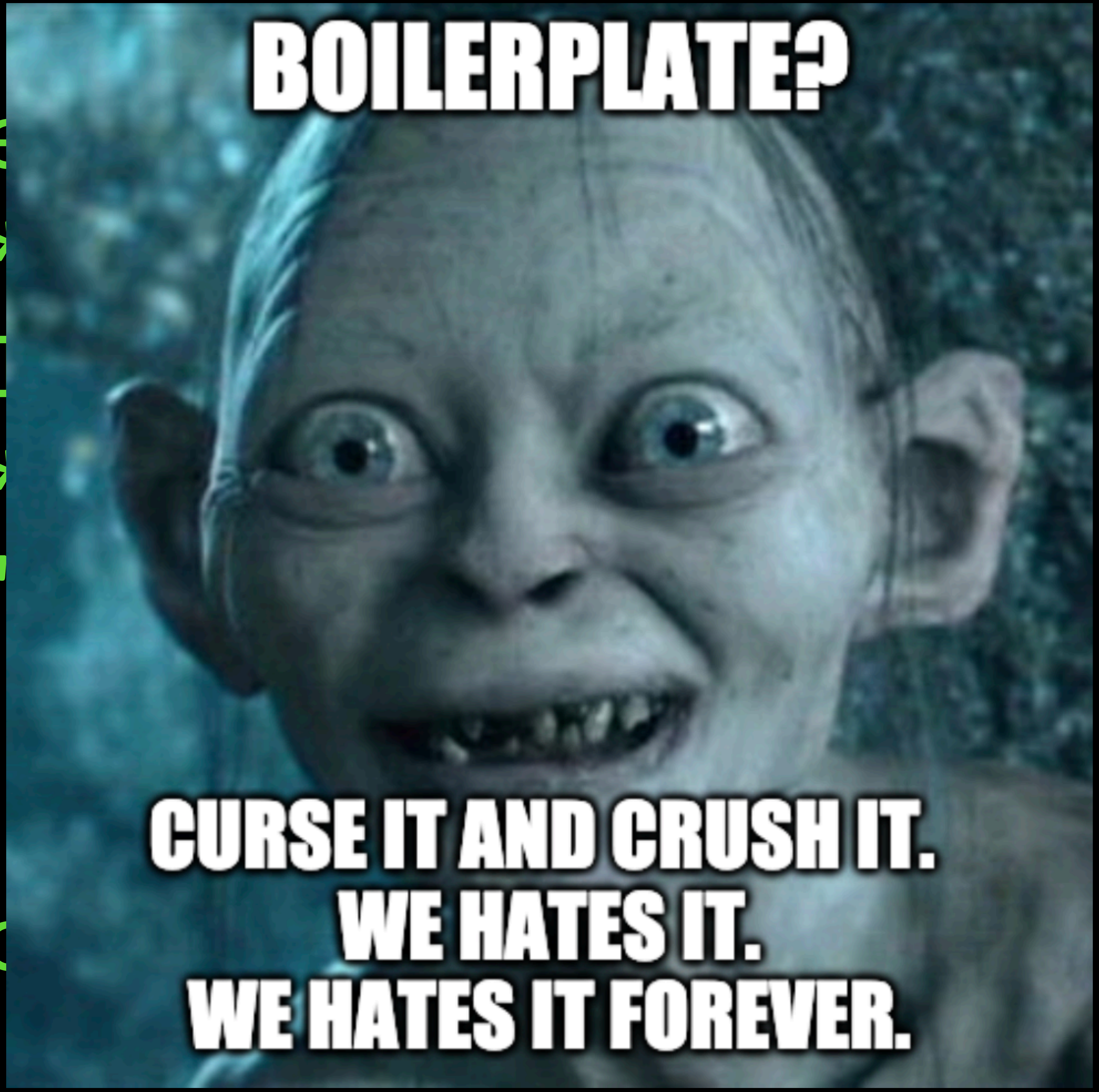
```
foo "Hello";

localized {
   my $*LOCALIZED = True;
   use Intl::UserLanguage;
   my $*LANGUAGE  = 'en';
   foo "Hello";
}

foo "Good-bye";
```

Localized Block

```
foo "Hello";

localized {
   my $*LOCALIZED = True;
   use Intl::UserLanguage;
   my $*LANGUAGE  = user-language;
   foo "Hello";
}

foo "Good-bye";
```

Localized Block

```
foo "Hello";

localize
    my $*
    use I                        ;
    my $*                        language;
    foo '
}

foo "Goo
```



Localized Block

```
sub localized (Block &block) {
    use Intl::UserLanguage;

    my $*LOCALIZED = True;
    my $*LANGUAGE = user-language;

    block();
}
```

Localized Block

```
say "hello";          # 'hello'

localized {
    say "hello";      # '¡Hola!'
    say "goodbye";    # '¡Adiós!'
}

say "bye";            # 'bye'
```

Localized Block

```
say "hello";          # 'hello'

localized {
    language 'ko';
    say "hello";      # '안녕!'
    say "goodbye"; # '잘 가!'
}

say "bye";            # 'bye'
```

Localized Block

```
sub localized (Block &block) {
    use Intl::UserLanguage;

    my $*LOCALIZED = True;
    my $*LANGUAGE = user-language;

    block();
}
```

Localized Block

```
sub localized (Block &block) {
    use Intl::UserLanguage;

    my $*LOCALIZED = True;
    my $*LANGUAGE = user-language;

    block();
}

sub language (Str $s) { $*LANGUAGE = $s }
```

Localized Block

```
unit module LocalizedBlocked;
sub localized (Block &block) is export {
    use Intl::UserLanguage;

    my $*LOCALIZED = True;
    my $*LANGUAGE = user-language;

    block();
}

sub language (Str $s) is export { $*LANGUAGE = $s }
```

Localized Block

```
say "hello";          # 'hello'

localized {
    language 'ko';
    say "hello";      # '안녕!'
    say "goodbye"; # '잘 가!'
}

say "bye";            # 'bye'
```

Localized Block

```
say "hello";         # 'hello'

localized {
    language 'de';
    say "hello";     # 'Hallo!'
    say "goodbye";   # 'Tchüss!'
}

say "bye";           # 'bye'
```

Localized Block

```
say "hello";          # 'hello'

localized {
    language 'chr';
    say "hello";      # 'ᎣᏏᏲ!'
    say "goodbye";    # 'ᏙᎾᏓᎬᎰᎢ!'
}

say "bye";            # 'bye'
```

Localized Block

```
unit module LocalizedBlocked;

#| Creates a localized environment to run code in
sub localized (
    Block &block #= Code to run with localized says
) is export {
    use Intl::UserLanguage;

    my $*LOCALIZED = True;
    my $*LANGUAGE = user-language;

    block();
}

#| Sets the language for a localized block
sub language (
    Str $s  #= Manually set the language of a localized block
) is export {
    warn "Useless use of language() outside of localized block"
        without $*LOCALIZED;
    $*LANGUAGE = $s
}

&say.wrap: sub ($s) {
  if   $*LOCALIZED { callwith translate $s }
  else             { callsame             }
}
```

Traits

Traits

Traits allow you to modify most things at compile time.

Traits

Traits allow you to modify most things at compile time.

```
class Foo is export {
  has $.thing   is rw;
  has $!private is built;
}
```

Traits

Traits allow you to modify most things at compile time.

```
class Foo is export {
  has $.thing   is rw;
  has $!private is built;
}
```

You might think that they're some very complex structure that's special cased in the compiler but …

Traits

Traits allow you to modify most things at compile time.

```
class Foo is export {
  has $.thing   is rw;
  has $!private is built;
}
```

You might think that they're some very complex structure that's special cased in the compiler but …

They're just subs.

Traits

Traits allow you to modify most things at compile time.

```
class Foo is export {
  has $.thing   is rw;
  has $!private is built;
}
```

You might think that they're some very complex structure that's special cased in the compiler but …

They're just subs.

Traits

So let's say we wanted to log access to a sub.

```
unit module SecretStuff;

sub get (|) { … }
```

Traits

So let's say we wanted to log access to a sub.

```
unit module SecretStuff;

sub get (|) is logged { … }
```

Traits

```
#| Logs access to any sub
multi sub trait_mod:<is> (
    Sub \r,     #= trait is applied to this
    :$logged!, #= name of trait
)
```

Traits

```
#| Logs access to any sub
multi sub trait_mod:<is> (
   Sub \r,     #= trait is applied to this
   :$logged!,  #= name of trait
) {
  r.wrap: sub (|args) {
    say "At {time}, called {r.name} with ", args;
    callsame
  }
}
```

Traits

```
#| Logs access to any sub
multi sub trait_mod:<is> (
   Sub \r,      #= trait is applied to this
   :$logged!,  #= name of trait
) {
  r.wrap: sub (|args) {
    say "At {time}, called {r.name} with ", args;
    callsame
  }
}
```

Traits

```
multi sub trait_mod:<is> (Sub \r, :$logged!) {
  r.wrap: sub (|args) {
    say "At {time}, called {r.name} with ", args;  callsame
  }
}
```

Traits

```
multi sub trait_mod:<is> (Sub \r, :$logged!) {
  r.wrap: sub (|args) {
    say "At {time}, called {r.name} with ", args;  callsame
  }
}
```

```
multi sub infix:<may-access> ($employee, $patient --> Bool) { … }
```

```
sub get-medical-data($patient, $employee) is logged {
   if $employee may-access* $patient {
      …
   }
}
```

Traits

```
multi sub trait_mod:<is> (Sub \r, :$logged!) {
  r.wrap: sub (|args) {
    say "At {time}, called {r.name} with ", args;  callsame
  }
}

multi sub infix:<may-access> ($employee, $patient --> Bool) { … }
```

```
sub get-medical-data($patient, $employee) is logged {
  if $employee may-access* $patient {
    …
  }
}


get-medical-data 'John', 'Dr. Jenkins';
get-medical-data 'Jane', 'Dr. Nguyen';
```

Traits

```
multi sub trait_mod:<is> (Sub \r, :$logged!) {
  r.wrap: sub (|args) {
    say "At {time}, called {r.name} with ", args;  callsame
  }
}

multi sub infix:<may-access> ($employee, $patient --> Bool) { … }

sub get-medical-data($patient, $employee) is logged {
  if $employee may-access* $patient {
    …
  }
}


get-medical-data 'John', 'Dr. Jenkins';
get-medical-data 'Jane', 'Dr. Nguyen';
```

```
At 1610736801, called get-medical-data with \("John", "Dr. Jenkins")
At 1610736801, called get-medical-data with \("Jane", "Dr. Nguyen")
```

Regexen / Tokens

Regexen / Tokens

```
grammar Foo {
  token TOP   { <alpha> <smile> }
  token smile {  ':-)'  |   😀    }
}
```

Regexen / Tokens

```
grammar Foo {
  token TOP   { <alpha> <smile> }
  token smile {  ':-)'  |   😀    }
}
```

The special syntax of <...> is technically just a method call that returns a `Match`.

Regexen / Tokens

```
grammar Foo {
  token TOP   { <alpha> <smile> }
  token smile {  ':-)'  |   😀    }
}
```

The special syntax of <...> is technically just a method call that returns a `Match`.

These can be declared outside of regexen/grammars to be used across multiple definitions.

Regexen / Tokens

Regexen / Tokens

```
my token happy {😀|😃|😄|😁|😆|😊|🙂}
my token sad   {😞|😟|🙁|☹️|😢|😭|😥}
my token flag  {  <[🇦..🇿]> ** 2  }
                   \x1F1E6 \x1F1FF
```

Regexen / Tokens

```
my token happy {😀|😃|😄|😁|😆|😊|🙂}
my token sad   {😞|😟|🙁|☹️|😢|😭|😥}
my token flag  {  <[🇦..🇿]> ** 2  }
                   \x1F1E6 \x1F1FF

sub describe($text) {
  say "Emotional" if $text ~~ /<happy> | <sad> /;
  say "Patriotic" if $text ~~ /<happy>   <flag>/;
}
```

Regexen / Tokens

```
my token happy {😀|😃|😄|😁|😆|😊|🙂}
my token sad   {😞|😟|🙁|☹️|😢|😭|😥}
my token flag  {  <[🇦..🇿]> ** 2  }
                  \x1F1E6 \x1F1FF

sub describe($text) {
  say "Emotional" if $text ~~ /<happy> | <sad> /;
  say "Patriotic" if $text ~~ /<happy>   <flag>/;
}

describe 'I got the job! 😁';       # Emotional
describe 'I failed the test 😢';    # Emotional
describe 'We won the gold! 😃🇺🇸';   # Patriotic
```

Regexen / Tokens

Tokens can also have code,
and can easily dictate how far to advance the token.

Regexen / Tokens

Tokens can also have code,
and can easily dictate how far to advance the token.

```
token foo {
    :my $advance = 0;

    {
        my $remainder = $/.orig.substr: $/.to;
        $advance = check $remainder;
    }

    . ** {$advance}
}
```

Regexen / Tokens

Tokens can also have code,
and can easily dictate how far to advance the token.

```
token foo {
    :my $advance = 0;

    <?{
        my $remainder = $/.orig.substr: $/.to;
        $advance = check $remainder;
    }>

    . ** {$advance}
}
```

Regexen / Tokens

Tokens can also have code,
and can easily dictate how far to advance the token.

```
token foo {
    :my $advance = 0;

    <?{
        my $remainder = $/.orig.substr: $/.to;
        $advance = check $remainder;
    }>

    . ** {$advance}
}
```

Don't forget the possibility of returning `0 but True` to make a `0` a truthy valid

Showcase

Modules that Just Work™

(and how)

silently

```
quietly {
  say "HAHAHAHA I'm a small child and
       make lots of noise in libraries";
  warn "There's a fire in the lobby!";
}
```

```
There's a fire in the lobby!
```

```
silently {
  say "HAHAHAHA I'm a small child and
       make lots of noise in libraries";
  warn "There's a fire in the lobby!";
}
```

```
[no output]
```

silently

```
quietly {
  say "HAHAHAHA I'm a small child and
       make lots of noise in libraries";
  warn "There's a fire in the lobby!";
}
```

```
There's a fire in the lobby!
```

```
silently {
  say "HAHAHAHA I'm a small child and
       make lots of noise in libraries";
  warn "There's a fire in the lobby!";
}
```

```
[no output]
```

```
sub silently(&code) is export {
    my $captured := Captured.new(my $*OUT, my $*ERR);
    &code();
    $captured
}
```

silently

```
quietly {
  say "HAHAHAHA I'm a small child and
       make lots of noise in libraries";
  warn "There's a fire in the lobby!";
}
```

```
There's a fire in the lobby!
```

```
silently {
  say "HAHAHAHA I'm a small child and
       make lots of noise in libraries";
  warn "There's a fire in the lobby!";
}
```

```
[no output]
```

Overwritten during building to a class that mimicks an IO::Handle, but saves output to be returned

```
sub silently(&code) is export {
    my $captured := Captured.new(my $*OUT, my $*ERR);
    &code();
    $captured
}
```

Cro

```
my $chat = Supplier.new;
get -> 'chat' {
    web-socket -> $incoming {
        supply {
            whenever $incoming -> $message {
                $chat.emit: await $message.body-text
            }
            whenever $chat -> $text {
                emit $text
            }
        }
    }
}
```

Cro

```
my $chat = Supplier.new;
get -> 'chat' {
sub
    web-socket -> $incoming {
    sub
        supply {
      control word
            whenever $incoming -> $message {
         control word
                $chat.emit: await $message.body-text
            }
            whenever $chat -> $text {
         control word
                emit $text
            }
        }
    }
}
```

Cro

introspection of the signature allows this to be equivalent to get 'chat', { … },
but look a bit more Raku-ish

```
my $chat = Supplier.new;
get -> 'chat' {
sub
    web-socket -> $incoming {
    sub
        supply {
      control word
            whenever $incoming -> $message {
         control word
                $chat.emit: await $message.body-text
            }
            whenever $chat -> $text {
         control word
                emit $text
            }
        }
    }
}
```

Cro

introspection of the signature allows this to be equivalent to get 'chat', { … }, but look a bit more Raku-ish

value passed in when called

```
my $chat = Supplier.new;
get -> 'chat' {
sub
    web-socket -> $incoming {
    sub
        supply {
    control word
            whenever $incoming -> $message {
        control word
                $chat.emit: await $message.body-text
            }
            whenever $chat -> $text {
        control word
                emit $text
            }
        }
    }
}
```

Red

```
model Post is rw {
    has Int $.id        is serial;
    has Str $.title     is unique;
    has Str $.body      is column;
    has Int $!author-id is referencing{   :model<Person>, :column<id>    };
    has     $.author    is relationship( *.author-id,     :model<Person> );
}

model Person is rw {
    has Int  $.id    is serial;
    has Str  $.name  is column;
    has      @.posts is relationship( *.author-id, :model<Post> );
}
```

Red

custom declarator

custom traits allow complex setup
to happen in the background

Because traits are subs, they can also accept
anything that a sub would as arguments:

```
model Post is rw {
    has Int $.id          is serial;
    has Str $.title       is unique;
    has Str $.body        is column;
    has Int $!author-id is referencing{   :model<Person>, :column<id>    };
    has     $.author    is relationship( *.author-id,      :model<Person> );
}

model Person is rw {
    has Int  $.id     is serial;
    has Str  $.name   is column;
    has      @.posts is relationship( *.author-id, :model<Post> );
}
```

Intl::Token::Number

```
my $text = "Houston is the most populous city in the U.S. state of Texas, fourth most populous city in the United States,
            most populous city in the Southern United States, as well as the sixth most populous in North America,
            with an estimated 2019 population of 2,320,268. Located in Southeast Texas near Galveston Bay and the Gulf
            of Mexico, it is the seat of Harris County and the principal city of the Greater Houston metropolitan area,
            which is the fifth most populous metropolitan statistical area in the United States and the second most
            populous in Texas after the Dallas–Fort Worth metroplex, with a population of 6,997,384 in 2018.

            Comprising a total area of 637.4 square miles (1,651 km2), Houston is the eighth most expansive city in the
            United States (including consolidated city-counties). It is the largest city in the United States by total
            area, whose government is not consolidated with that of a county, parish or borough. Though primarily in
            Harris County, small portions of the city extend into Fort Bend and Montgomery counties, bordering other
            principal communities of Greater Houston such as Sugar Land and The Woodlands.

            Houston's characteristic subtropical humidity often results in a higher apparent temperature, and summer
            mornings average over 90% relative humidity";
```

Intl::Token::Number

```
my $text = "Houston is the most populous city in the U.S. state of Texas, fourth most populous city in the United States,
            most populous city in the Southern United States, as well as the sixth most populous in North America,
            with an estimated 2019 population of 2,320,268. Located in Southeast Texas near Galveston Bay and the Gulf
            of Mexico, it is the seat of Harris County and the principal city of the Greater Houston metropolitan area,
            which is the fifth most populous metropolitan statistical area in the United States and the second most
            populous in Texas after the Dallas–Fort Worth metroplex, with a population of 6,997,384 in 2018.

            Comprising a total area of 637.4 square miles (1,651 km2), Houston is the eighth most expansive city in the
            United States (including consolidated city-counties). It is the largest city in the United States by total
            area, whose government is not consolidated with that of a county, parish or borough. Though primarily in
            Harris County, small portions of the city extend into Fort Bend and Montgomery counties, bordering other
            principal communities of Greater Houston such as Sugar Land and The Woodlands.

            Houston's characteristic subtropical humidity often results in a higher apparent temperature, and summer
            mornings average over 90% relative humidity";
```

```
for $text.match: /<local-number>/, :g -> $\ {
  say "{~$<local-number>} is equal to {+$<local-number>";
}
```

Intl::Token::Number

```
my $text = "Houston is the most populous city in the U.S. state of Texas, fourth most populous city in the United States,
            most populous city in the Southern United States, as well as the sixth most populous in North America,
            with an estimated 2019 population of 2,320,268. Located in Southeast Texas near Galveston Bay and the Gulf
            of Mexico, it is the seat of Harris County and the principal city of the Greater Houston metropolitan area,
            which is the fifth most populous metropolitan statistical area in the United States and the second most
            populous in Texas after the Dallas–Fort Worth metroplex, with a population of 6,997,384 in 2018.

            Comprising a total area of 637.4 square miles (1,651 km2), Houston is the eighth most expansive city in the
            United States (including consolidated city-counties). It is the largest city in the United States by total
            area, whose government is not consolidated with that of a county, parish or borough. Though primarily in
            Harris County, small portions of the city extend into Fort Bend and Montgomery counties, bordering other
            principal communities of Greater Houston such as Sugar Land and The Woodlands.

            Houston's characteristic subtropical humidity often results in a higher apparent temperature, and summer
            mornings average over 90% relative humidity";
```

```
for $text.match: /<local-number>/, :g -> $\ {
  say "{~$<local-number>} is equal to {+$<local-number>";
}
```

Intl::Token::Number

```
my $text = "Houston is the most populous city in the U.S. state of Texas, fourth most populous city in the United States,
            most populous city in the Southern United States, as well as the sixth most populous in North America,
            with an estimated 2019 population of 2,320,268. Located in Southeast Texas near Galveston Bay and the Gulf
            of Mexico, it is the seat of Harris County and the principal city of the Greater Houston metropolitan area,
            which is the fifth most populous metropolitan statistical area in the United States and the second most
            populous in Texas after the Dallas–Fort Worth metroplex, with a population of 6,997,384 in 2018.

            Comprising a total area of 637.4 square miles (1,651 km2), Houston is the eighth most expansive city in the
            United States (including consolidated city-counties). It is the largest city in the United States by total
            area, whose government is not consolidated with that of a county, parish or borough. Though primarily in
            Harris County, small portions of the city extend into Fort Bend and Montgomery counties, bordering other
            principal communities of Greater Houston such as Sugar Land and The Woodlands.

            Houston's characteristic subtropical humidity often results in a higher apparent temperature, and summer
            mornings average over 90% relative humidity";
```

```
for $text.match: /<local-number>/, :g -> $\ {
  say "{~$<local-number>} is equal to {+$<local-number>";
}
```

```
2019 is equal to 2019
2,320,268 is equal to 2320268
6,997,384 is equal to 6997384
2018 is equal to 2018
637.4 is equal to 637.4
1,651 is equal to 1651
2 is equal to 2
90% is equal to 0.9
```

Intl::Token::Number

```
my $text = "Houston is the most populous city in the U.S. state of Texas, fourth most populous city in the United States,
            most populous city in the Southern United States, as well as the sixth most populous in North America,
            with an estimated 2019 population of 2,320,268. Located in Southeast Texas near Galveston Bay and the Gulf
            of Mexico, it is the seat of Harris County and the principal city of the Greater Houston metropolitan area,
            which is the fifth most populous metropolitan statistical area in the United States and the second most
            populous in Texas after the Dallas–Fort Worth metroplex, with a population of 6,997,384 in 2018.

            Comprising a total area of 637.4 square miles (1,651 km2), Houston is the eighth most expansive city in the
            United States (including consolidated city-counties). It is the largest city in the United States by total
            area, whose government is not consolidated with that of a county, parish or borough. Though primarily in
            Harris County, small portions of the city extend into Fort Bend and Montgomery counties, bordering other
            principal communities of Greater Houston such as Sugar Land and The Woodlands.

            Houston's characteristic subtropical humidity often results in a higher apparent temperature, and summer
            mornings average over 90% relative humidity";
```

token wrapped by a method

```
for $text.match: /<local-number>/, :g -> $\ {
  say "{~$<local-number>} is equal to {+$<local-number>";
}
```

```
2019 is equal to 2019
2,320,268 is equal to 2320268
6,997,384 is equal to 6997384
2018 is equal to 2018
637.4 is equal to 637.4
1,651 is equal to 1651
2 is equal to 2
90% is equal to 0.9
```

Intl::Token::Number

```
my $text = "Houston is the most populous city in the U.S. state of Texas, fourth most populous city in the United States,
            most populous city in the Southern United States, as well as the sixth most populous in North America,
            with an estimated 2019 population of 2,320,268. Located in Southeast Texas near Galveston Bay and the Gulf
            of Mexico, it is the seat of Harris County and the principal city of the Greater Houston metropolitan area,
            which is the fifth most populous metropolitan statistical area in the United States and the second most
            populous in Texas after the Dallas–Fort Worth metroplex, with a population of 6,997,384 in 2018.

            Comprising a total area of 637.4 square miles (1,651 km2), Houston is the eighth most expansive city in the
            United States (including consolidated city-counties). It is the largest city in the United States by total
            area, whose government is not consolidated with that of a county, parish or borough. Though primarily in
            Harris County, small portions of the city extend into Fort Bend and Montgomery counties, bordering other
            principal communities of Greater Houston such as Sugar Land and The Woodlands.

            Houston's characteristic subtropical humidity often results in a higher apparent temperature, and summer
            mornings average over 90% relative humidity";
```

token wrapped by a method

```
for $text.match: /<local-number>/, :g -> $\ {
  say "{~$<local-number>} is equal to {+$<local-number>";
}
```

wrapping mixes in a role with a .Numeric method

```
2019 is equal to 2019
2,320,268 is equal to 2320268
6,997,384 is equal to 6997384
2018 is equal to 2018
637.4 is equal to 637.4
1,651 is equal to 1651
2 is equal to 2
90% is equal to 0.9
```

Test::Inline

```
unit module Rectangle;

use Test::Inline;

has Point $.a; # bottom left
has Point $.b; # top right

sub calculate-area($x, $y) {     $x * $y }
sub distance(      $a, $b) { abs $a - $b }

method area {
  calculate-area
    distance($!a.x, $!b.x),
    distance($!a.y, $!b.y)
}

method overlap(Rectangle $other) { ... }

sub t-distance is test {
  use Test;
  is distance( 2, 4), 2, "+/+";
  is distance(-2, 4), 6, "-/+";
  is distance(-2,-1), 1, "-/-";
}

sub t-area is test {
  use Test;
  is ……… , "area A";
  is ……… , "area B";
}
```

Test::Inline

```
unit module Rectangle;

use Test::Inline;

has Point $.a; # bottom left
has Point $.b; # top right

sub calculate-area($x, $y) {     $x * $y }
sub distance(      $a, $b) { abs $a - $b }

method area {
  calculate-area
    distance($!a.x, $!b.x),
    distance($!a.y, $!b.y)
}

method overlap(Rectangle $other) { ... }

sub t-distance is test {
  use Test;
  is distance( 2, 4), 2, "+/+";
  is distance(-2, 4), 6, "-/+";
  is distance(-2,-1), 1, "-/-";
}

sub t-area is test {
  use Test;
  is ……… , "area A";
  is ……… , "area B";
}
```

```
use Test;
use Test::Inline, :testing;

use Rectangle;

my $r = Rectangle.new:
  a => Point.new(2,3),
  b => Point.new(5,6);

is $r.a.x, 2, "x";
is $r.b.y, 6, "y";

inline-testing;

done-testing;
```

Test::Inline

```
unit module Rectangle;

use Test::Inline;

has Point $.a; # bottom left
has Point $.b; # top right

sub calculate-area($x, $y) {     $x * $y }
sub distance(      $a, $b) { abs $a - $b }

method area {
  calculate-area
    distance($!a.x, $!b.x),
    distance($!a.y, $!b.y)
}

method overlap(Rectangle $other) { ... }

sub t-distance is test {
  use Test;
  is distance( 2, 4), 2, "+/+";
  is distance(-2, 4), 6, "-/+";
  is distance(-2,-1), 1, "-/-";
}

sub t-area is test {
  use Test;
  is ……… , "area A";
  is ……… , "area B";
}
```

```
use Test;
use Test::Inline, :testing;

use Rectangle;

my $r = Rectangle.new:
  a => Point.new(2,3),
  b => Point.new(5,6);

is $r.a.x, 2, "x";
is $r.b.y, 6, "y";

inline-testing;

done-testing;
```

```
ok 1 - x
ok 2 - y
            is 1 - +/+
            is 2 - -/+
            is 3 - -/-
            1..3
        ok 1 - sub t-distance
            is 1 - area A
            is 2 - area B
            1..2
        ok 1 - sub t-area
        1..2
    ok 1 - Package Rectangle
    1..1
ok 3 - Inline testing
```

Test::Inline

```
unit module Rectangle;

use Test::Inline;

has Point $.a; # bottom left
has Point $.b; # top right

sub calculate-area($x, $y) {     $x * $y }
sub distance(      $a, $b) { abs $a - $b }

method area {
  calculate-area
    distance($!a.x, $!b.x),
    distance($!a.y, $!b.y)
}

method overlap(Rectangle $other) { ... }

sub t-distance is test {
  use Test;
  is distance( 2, 4), 2, "+/+";
  is distance(-2, 4), 6, "-/+";
  is distance(-2,-1), 1, "-/-";
}

sub t-area is test {
  use Test;
  is ……… , "area A";
  is ……… , "area B";
}
```

```
use Test;
use Test::Inline, :testing;

use Rectangle;

my $r = Rectangle.new:
  a => Point.new(2,3),
  b => Point.new(5,6);

is $r.a.x, 2, "x";
is $r.b.y, 6, "y";

inline-testing;

done-testing;


ok 1 - x
ok 2 - y
            is 1 - +/+
            is 2 - -/+
            is 3 - -/-
            1..3
        ok 1 - sub t-distance
            is 1 - area A
            is 2 - area B
            1..2
        ok 1 - sub t-area
        1..2
    ok 1 - Package Rectangle
    1..1
ok 3 - Inline testing
```

```
unit module Inline;

my Sub @tests;

#| Marks a sub as being for internal test purposes
multi sub trait_mod:<is>(Sub $sub, :$test!) is export {
    @tests.push: $sub if $test;
}

#| Calls all subs marked as 'is test' in loaded modules
sub inline-testing is export(:testing) {
    use Test;

    # Provided by the Test module
    subtest {
        for @tests.categorize(*.package.^name).sort(*.key)
         -> (:key($package), :value(@subs)) {

            subtest {
                for @subs.sort(*.name) -> &test {
                    subtest { test }, "sub {&test.name}";
                }
            }, "Package $package";
        }
    }, "Inline testing";
}
```

Intl::LanguageTag

```
class LanguageTag {
   method new (Str() $tag) {
       self.bless: …
   }
   method Str($?CLASS:D:) {
       # reverse of the above
   }
}
```

```
sub foo (LanguageTag() $x) {
   say $x.region
}

foo 'en-US' # errors!
```

Intl::LanguageTag

```
class LanguageTag {
   method new (Str() $tag) {
       self.bless: …
   }
   method Str($?CLASS:D:) {
       # reverse of the above
   }
   method COERCE(Str $tag) {
      self.new: $tag
   }
}
```

```
sub foo (LanguageTag() $x) {
   say $x.region
}

foo 'en-US' # '[Region:US]'
```

Here there be dragons

Here there be dragonflies?

Slang::SQL

```
my $*DB = DBIish.connect('SQLite', :database<sqlite.sqlite3>);

sql drop table if exists stuff; #runs 'drop table if exists stuff';

sql create table if not exists stuff (
      id  integer,
      sid varchar(32)
    );

for ^5 {
  sql insert into stuff (id, sid)
    values (?, ?); with ($_, ('A'..'Z').pick(16).join);
}

sql select * from stuff order by id asc; do -> $row {
  FIRST "{$*STATEMENT}id\tsid".say;
  "{$row<id>}\t{$row<sid>}".say;
};
```

Slang::SQL

```
my $*DB = DBIish.connect('SQLite', :database<sqlite.sqlite3>);

sql drop table if exists stuff; #runs 'drop table if exists stuff';

sql create table if not exists stuff (
      id  integer,
      sid varchar(32)
    );

for ^5 {
  sql insert into stuff (id, sid)
    values (?, ?); with ($_, ('A'..'Z').pick(16).join);
}

sql select * from stuff order by id asc; do -> $row {
  FIRST "{$*STATEMENT}id\tsid".say;
  "{$row<id>}\t{$row<sid>}".say;
};
```

Because of the flexibility inherent in Raku, the primary use for slangs will likely be creating special quoting languages.

Because of the flexibility inherent in Raku, the primary use for slangs will likely be creating special quoting languages.

Effectively, these will be akin to

Because of the flexibility inherent in Raku, the primary use for slangs will likely be creating special quoting languages.

Effectively, these will be akin to

Because of the flexibility inherent in Raku, the primary use for slangs will likely be creating special quoting languages.

Effectively, these will be akin to

```
sub circumfix:<sql ;> { … } # sql
sub circumfix:<bx/ /> { … } # binex
```

Because of the flexibility inherent in Raku, the primary use for slangs will likely be creating special quoting languages.

Effectively, these will be akin to

```
sub circumfix:<sql ;> { … } # sql
sub circumfix:<bx/ /> { … } # binex
```

Except that they will allow the circumfixed content to behave differently, not unlike how `rx/…/` or `Q:…:` works today.

Because of the flexibility inherent in Raku, the primary use for slangs will likely be creating special quoting languages.

Effectively, these will be akin to

```
sub circumfix:<sql ;> { … } # sql
sub circumfix:<bx/ /> { … } # binex
```

Except that they will allow the circumfixed content to behave differently, not unlike how `rx/…/` or `Q:…:` works today.

As RakuAST is committed to core, it will be even easier to integrate them at the same level that Q or Regex is in Raku.

All this said …

All this said …

It is possible to mimic quite a few bits of the main Raku language without needing to jump into slangs.

All this said …

It is possible to mimic quite a few bits of the main Raku language without needing to jump into slangs.

So, we can avoid the realm of dragon(flie)s and still do some surprisingly cool things, while functioning in utterly unsurprising ways for our users.

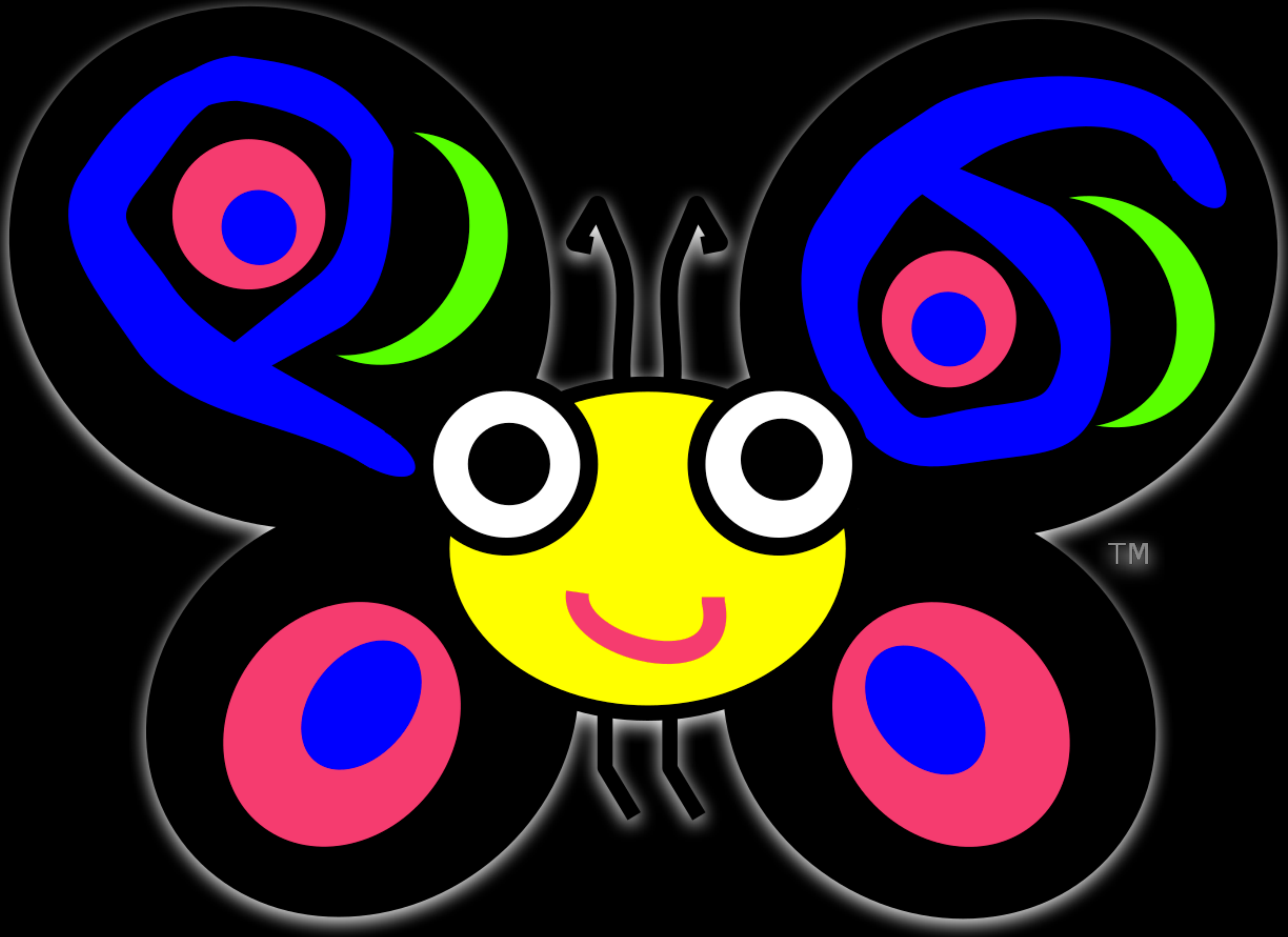
Module Development Checklist

Module Development Checklist

1. Think how the **user** would want to use your module.

Module Development Checklist

1. Think how the **user** would want to use your module.
    - Avoid boilerplate

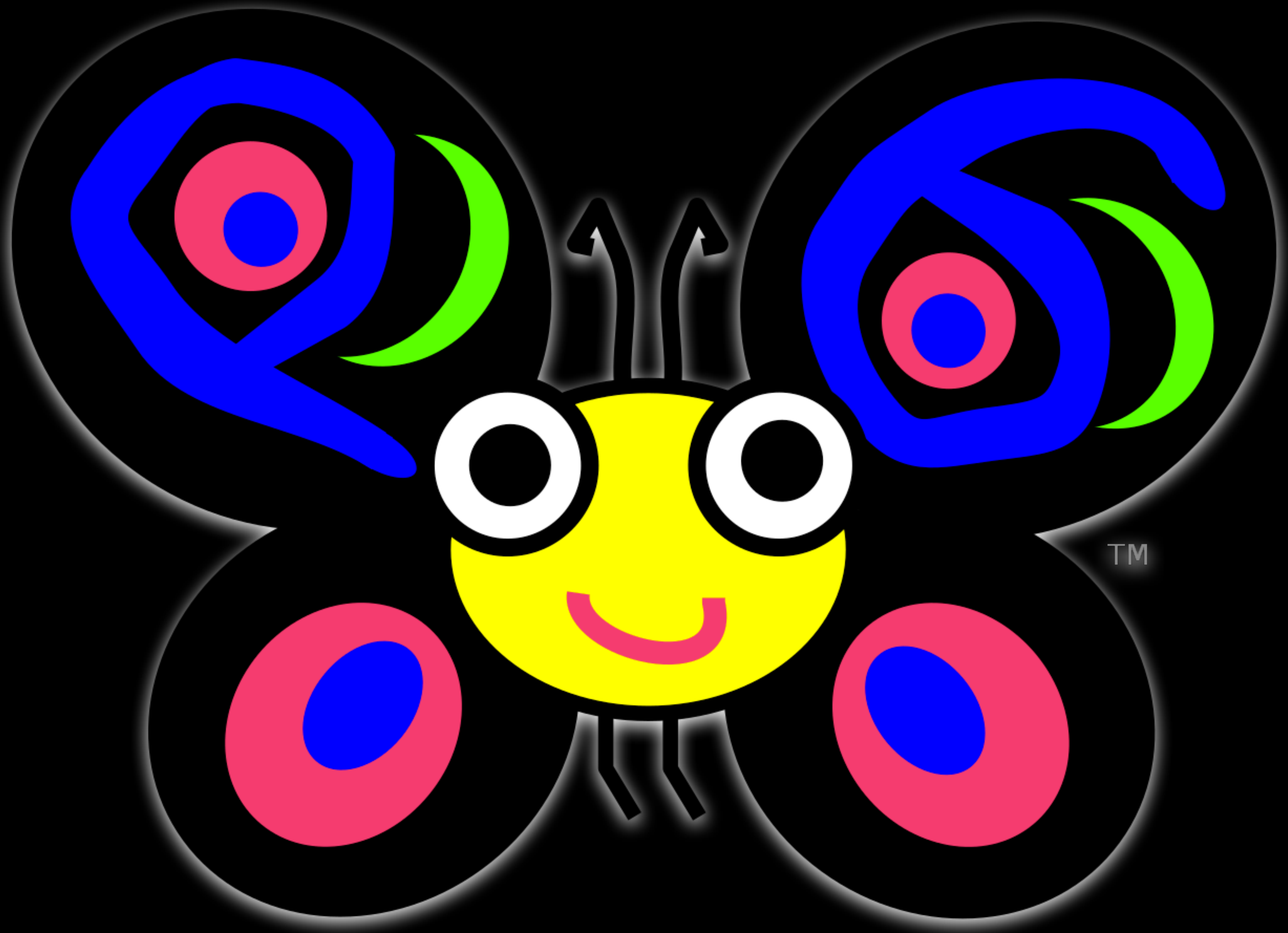


Module Development Checklist

1. Think how the **user** would want to use your module.
    - Avoid boilerplate
    - …while still providing options

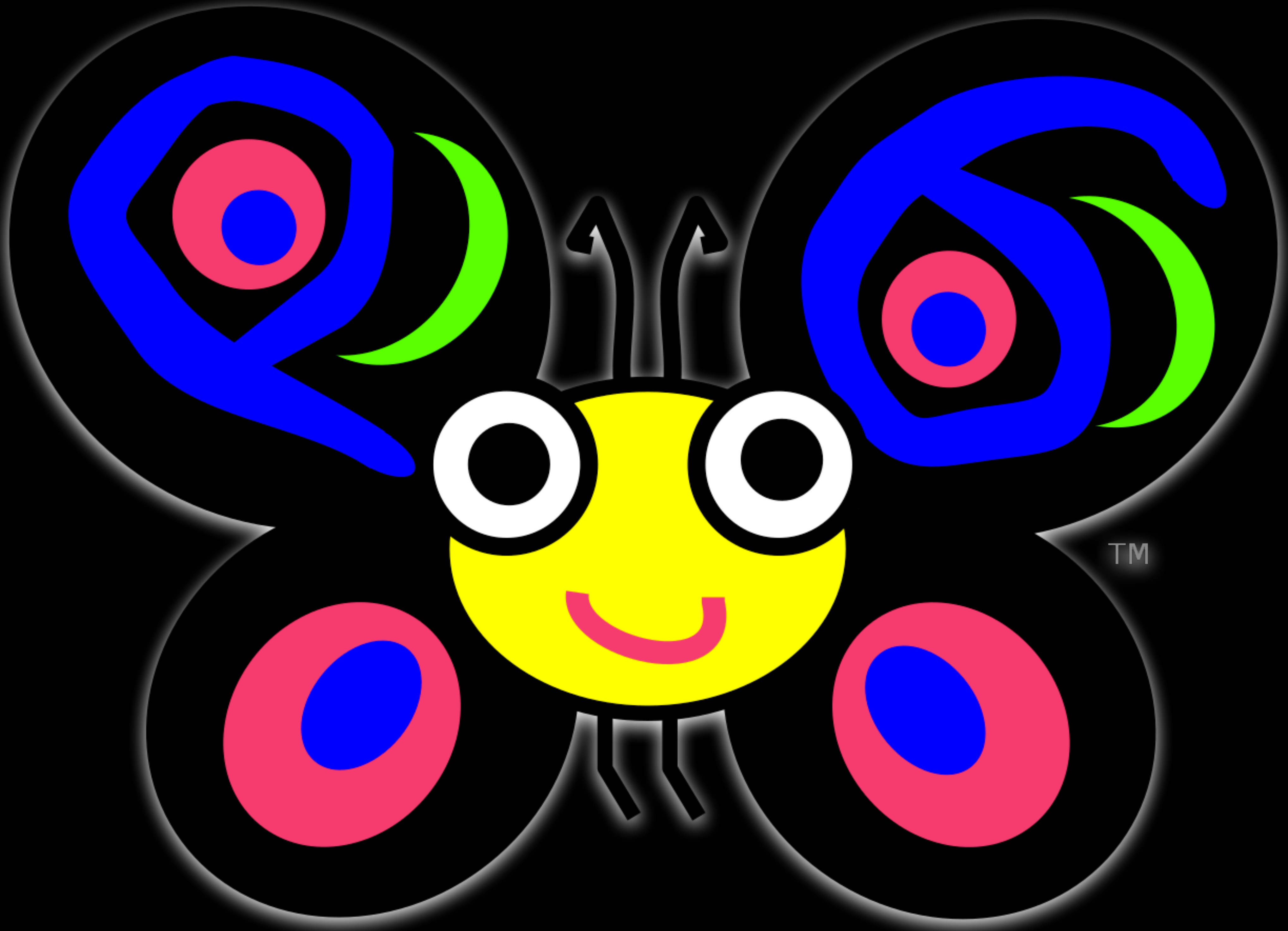
Module Development Checklist

1. Think how the **user** would want to use your module.
    - Avoid boilerplate
    - …while still providing options
2. Avoid putting your module in a bubble



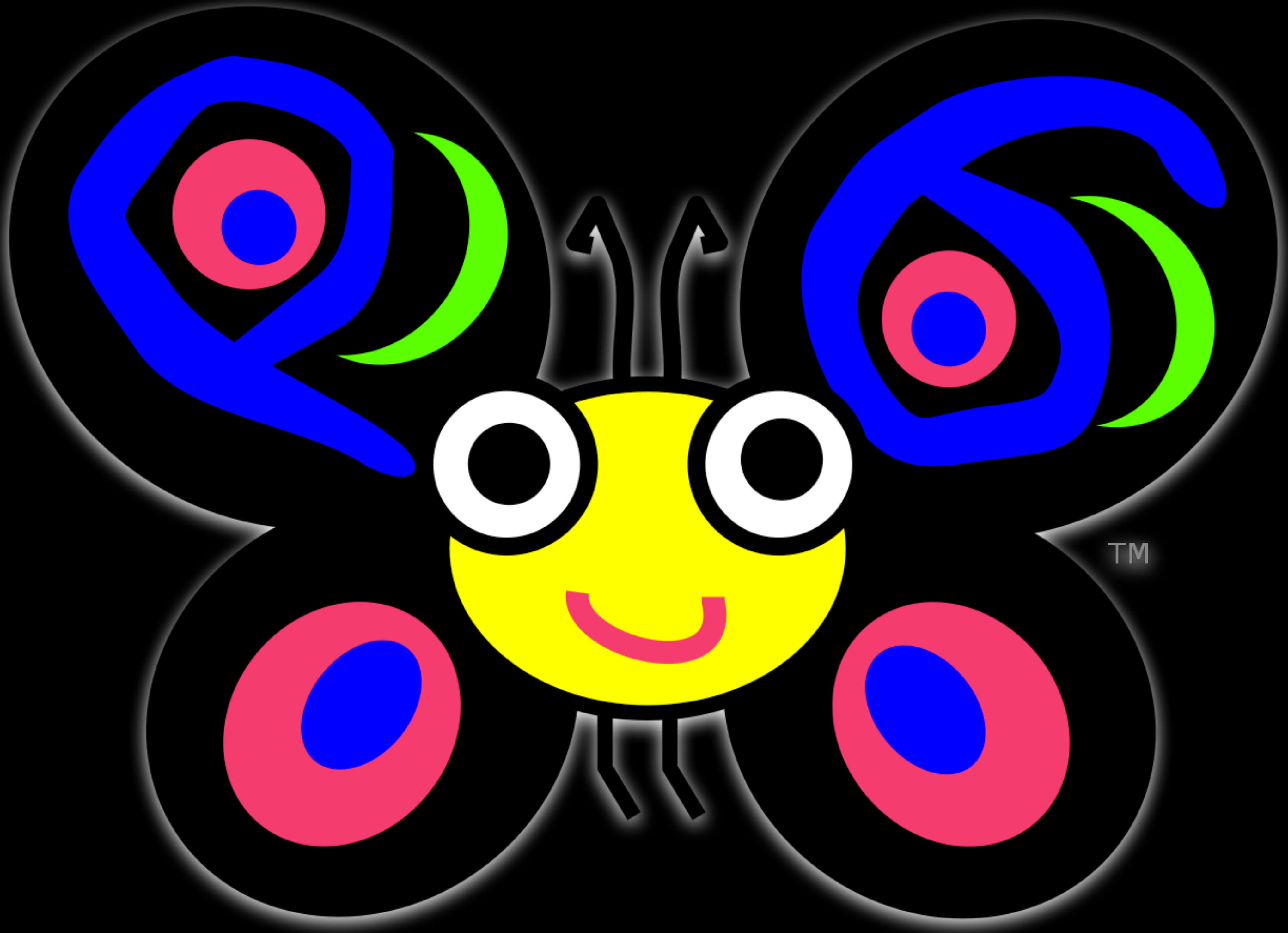
Module Development Checklist

1. Think how the **user** would want to use your module.
    - Avoid boilerplate
    - …while still providing options
2. Avoid putting your module in a bubble
    - Provide logical **`ACCEPT`**, **`COERCE`**, **`Str`**, and **`Numeric`** methods.



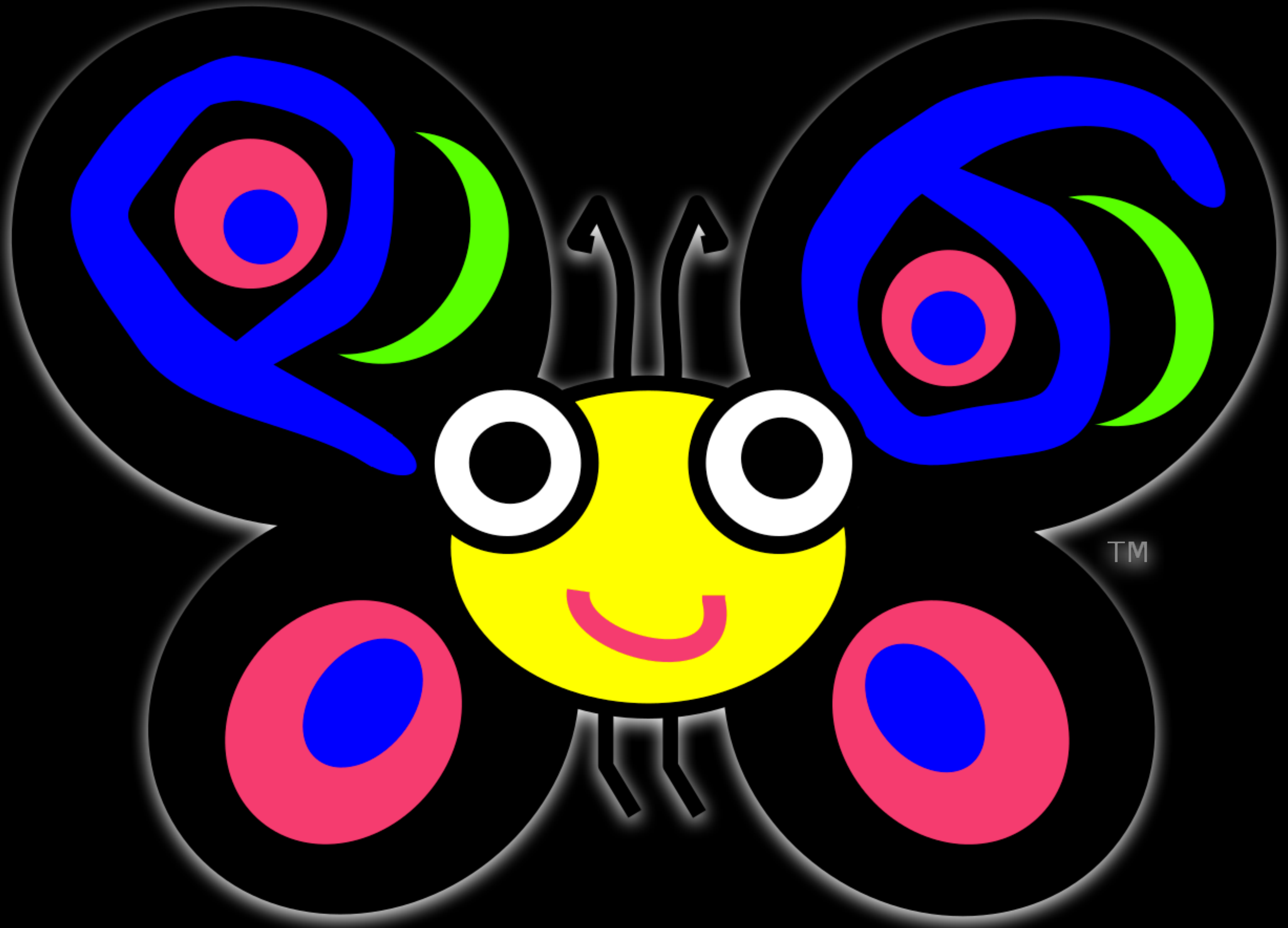
Module Development Checklist

1. Think how the **user** would want to use your module.
   - Avoid boilerplate
   - …while still providing options
2. Avoid putting your module in a bubble
   - Provide logical **`ACCEPT`**, **`COERCE`**, **`Str`**, and **`Numeric`** methods.
   - Go beyond a mere class/sub



Module Development Checklist

1. Think how the **user** would want to use your module.
    - Avoid boilerplate
    - …while still providing options
2. Avoid putting your module in a bubble
    - Provide logical `ACCEPT`, `COERCE`, `Str`, and `Numeric` methods.
    - Go beyond a mere class/sub
    - Support multiple paradigms/use cases by embracing Raku's native flexibility

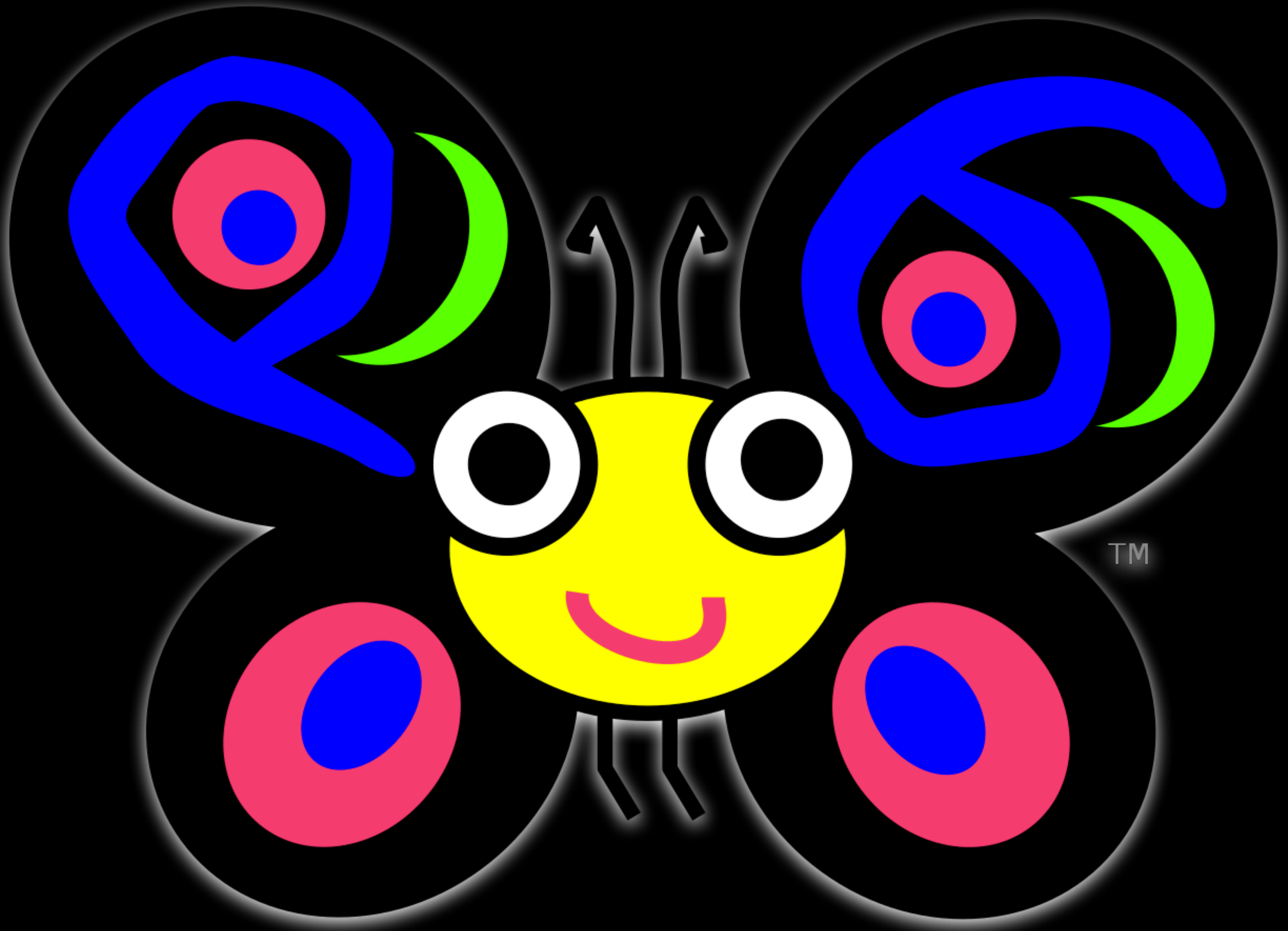


Module Development Checklist

1. Think how the **user** would want to use your module.
    - Avoid boilerplate
    - …while still providing options
2. Avoid putting your module in a bubble
    - Provide logical `ACCEPT`, `COERCE`, `Str`, and `Numeric` methods.
    - Go beyond a mere class/sub
    - Support multiple paradigms/use cases by embracing Raku's native flexibility
3. Document (whole ’nother talk)

Module Development Checklist

1. Think how the **user** would want to use your module.
   - Avoid boilerplate
   - …while still providing options
2. Avoid putting your module in a bubble
   - Provide logical **`ACCEPT`**, **`COERCE`**, **`Str`**, and **`Numeric`** methods.
   - Go beyond a mere class/sub
   - Support multiple paradigms/use cases by embracing Raku's native flexibility
3. Document (whole ’nother talk)
4. **Surprise the user with Raku-ish mundanity**



Any questions?

Or after the presentation:
guifa on #raku
alabamenhu on github
mateu@softastur.org